Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Integración del brazo robot IRB120 en entorno ROS - MATLAB

Autor: José Manuel Gómez Cuadrado

Tutor/es: Rafael Barea Navarro

2017

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Integración del brazo robot IRB120 en entorno ROS - MATLAB

Autor: José Manuel Gómez Cuadrado.

Director: Rafael Barea Navarro.

Tribunal:

Presidente: Elena López Guillén.

Vocal 1º: José Manuel Rodríguez Ascariz.

Vocal 2º: Rafael Barea Navarro.

Calificación: .....

Fecha: .....

Agradecimientos

Me gustaría agradecer a mi familia por ser constantes en mi aprendizaje y darme la oportunidad de estudiar este grado y a mi novia por apoyarme siempre.

Agradecer a todos mis compañeros que a lo largo de toda la carrera me han ayudado a superarla día a día.

A Rafael Barea por ser mi tutor y ayudarme con cada duda o con cada problema que me ha podido surgir.

Finalmente quería agradecer a Javier Moriano por sacar tiempo para ayudarme tanto con esta continuación a su proyecto.

Resumen

Este proyecto usa el entorno ROS (Robot Operating System) para desarrollar el control del brazo robot IRB 120 y su implementación en el entorno de trabajo MATLAB. Se explicará la creación del modelo del robot, la planificación de trayectorias y la comunicación con dicho robot.

Palabras clave: ROS, IRB-120, MATLAB.

Abstract

This project uses the ROS (Robot Operating System) environment for developing the control of the IRB 120 robotic arm and its implementation in the MATLAB working environment. It will explain the creation of the robot model, the planning of trajectories and the communication with that robot.

Keywords: ROS, IRB-120, MATLAB.

Resumen extendido

Este proyecto propone una solución para el control de la trayectoria del brazo robot ABB IRB120 a través del entorno de trabajo MATLAB.

Los principales objetivos son:

1. Poder recibir el estado en el que se encuentra el robot.

2. Mandar distintas posiciones al brazo robot a través de MATLAB para que las alcance.

Estos objetivos lo que permiten es un estudio de la comunicación entre ROS y MATLAB. Normalmente las empresas tienen su propio software para el manejo de los robots. Con esta comunicación lo que se quiere es poder abrir las puertas a nuevas formas de comunicación sin necesidad de usar distintos software.

Es decir, simplemente con recoger la información en la que se encuentran las distintas partes del robot se puede saber dónde se encuentra y así saber si la posición a la que se le quiere mandar es posible o no.

Además, se comprenderá cómo se trabaja en ROS, una herramienta cada día más usada y más útil. Con ROS se puede conseguir unir el brazo robot y MATLAB, es decir, es el puente que hace posible que esta comunicación se lleve a cabo.

Todo se realiza con mensajes que se mandan (desde MATLAB) y que recibe el brazo para saber hasta qué punto se debe desplazar. Se utiliza RVIZ para la realización de la trayectoria. Sabiendo el punto inicial y el final del brazo, RVIZ es capaz de calcular una trayectoria para que la realice el brazo.

Contenidos

Lista de figuras

Lista de tablas

Capítulo 1

Introducción

Día a día se crece tecnológicamente y gracias a ello, se van incorporando diferentes tipos de robots a las industrias. Esto es debido a que son capaces de llevar a cabo tareas en condiciones muy extremas o en lugares peligrosos con grandes resultados. Estos robots pueden ser de varios tipos, desde brazos robots a robots humanoides.

Actualmente las empresas de gran magnitud tienen una gran unión con la robótica debido a que pueden realizar actividades repetitivas que para los seres humanos podrían resultarles tediosas. La robótica se ha introducido a nivel industrial con fuerza, creando nuevos mecanismos de producción y automatización. El robot industrial se basa en la unión de una estructura mecánica y otra de electrónica para el sistema de control.

Figura 1: Fabricación en serie automatizada.

Un robot industrial genera una serie de movimientos que como tales se especifican y diseñan para una función especificada o conocida. Es decir, tendrán movimientos y operaciones concretas a ejecutar. Los mecánicos de cada robot industrial como se menciona anteriormente siempre tendrán una parte de estructura mecánica y otra de electrónica para el control y el manejo de cualquier necesidad a ejecutar.

Cuando se tiene un sistema de fabricación flexible en una empresa, a través del empleo de robots, se deja la producción en masa de un producto único y se pasa a la producción flexible diversificada de pequeños lotes, en la que la modificación de un producto se realiza sin alterar la estructura física de la máquina, sino más bien modificando tan sólo el programa que se debe ejecutar.

Actualmente, los robots industriales no toman decisiones en función de parámetros o factores que cambian de forma durante su funcionamiento, sin embargo los recientes progresos en los campos de la inteligencia artificial, del aprendizaje y del reconocimiento de formas permiten a los robots industriales obtener información de sus entornos y adaptar su comportamiento a las modificaciones del mismo, sin necesidad de intervención de un operario.

Figura 2: Brazo robot en un laboratorio de pruebas.

Cada robot industrial está basado en la red lógica, que a su vez es una de las más importantes. Es decir, el cerebro del robot es la programación con sus restricciones y condiciones para su debido funcionamiento, ya que estos deben cumplir una función determinada impuesta o necesaria para las empresas industriales. Cada componente mecánico es necesario, como de necesario es que tenga el mínimo error de estructura para su mejor funcionamiento.

El control de estos diferentes robots se lleva a cabo a través de diferentes tipos de microprocesadores y microcontroladores. Además, cada diseñador tiene su propio lenguaje de programación o sus propias herramientas con las que lleva a cabo la comunicación entre el robot y el usuario.

Figura 3: Brazos robot ABB.

Haciendo referencia en el campo de los brazos robots, casi todos los brazos realizan las mismas tareas, ya sea el montaje de un coche, la colocación de ciertos productos para su almacenaje… Es decir, realizan una rutina repetitiva. Estas tareas suelen estar programadas a través de diferentes trayectorias, con las cuales se controla la manejabilidad del brazo robot.

Para realizar este tipo de comunicación, (la programación del brazo robot) muchas veces hay que acoplarse al gusto del diseñador. Por ello, en este proyecto se intenta globalizar un poco los diferentes software. El usuario simplemente mandando una posición al brazo a través de MATLAB va a ser capaz de realizar diferentes trayectorias. Es decir, usando ROS y MATLAB se puede crear un puente de comunicación entre el brazo robot y el usuario.

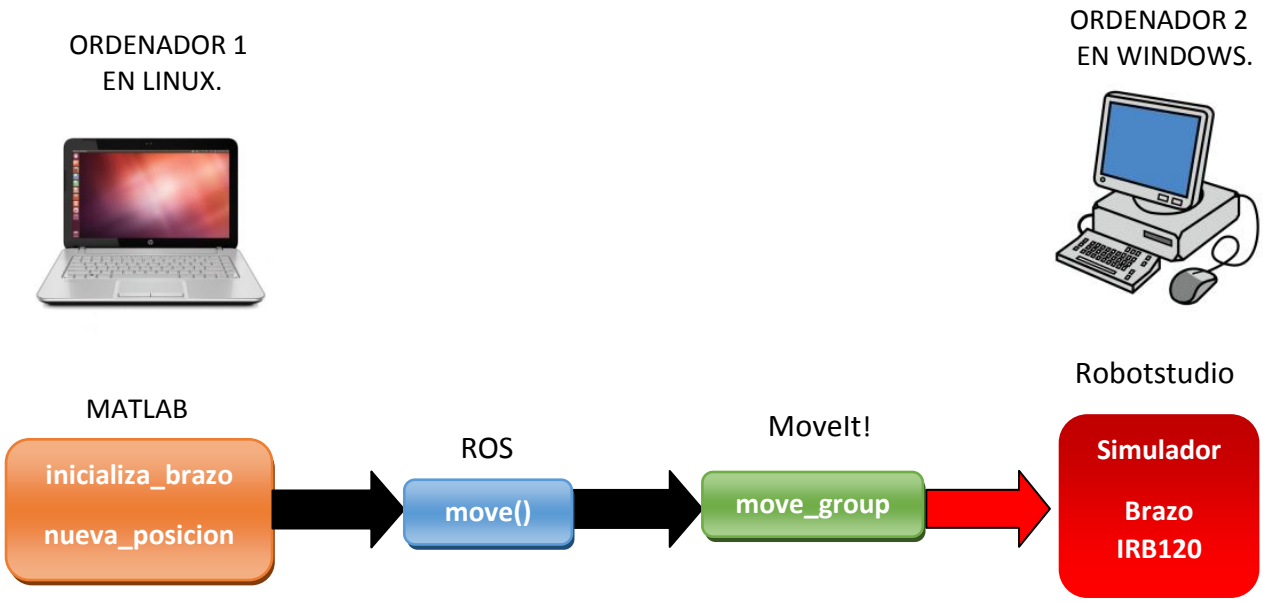
Figura 4: Brazo robot realizando una tarea de gran precisión.

Capítulo 2

Arquitectura general del sistema

2.1 Diagrama de desarrollo

Para el desarrollo de este proyecto, son necesarias varias herramientas. Se utilizan dos ordenadores con distintos sistemas operativos. En el ordenador con Linux se lleva a cabo el puente de comunicación con ROS, la inicialización de MoveIt y la ejecución de MATLAB. En el ordenador con Windows se realiza la ejecución de Robotstudio y la simulación del brazo robot en el mismo. Por lo tanto, las herramientas son las siguientes:



2.2 MATLAB

Es una herramienta muy utilizada en muchos sistemas de aprendizaje. Esto se debe a que MATLAB es una plataforma que está optimizada para resolver problemas de ingeniería y científicos. Los gráficos integrados facilitan mucho la visualización de los datos y la obtención de información a partir de ellos. Hay una gran diversidad de librerías preinstaladas que permiten trabajar inmediatamente con algoritmos para su dominio o ver ejemplos de aplicaciones que se quieran realizar.

Además, permite crear archivos .m que facilitan a la hora de realizar las simulaciones, ya que realiza una corrección línea a línea del código y permite tanto simular el código entero del archivo como manipular posteriormente órdenes incluidas en estos archivos o variables.

Para realizar la comunicación con ROS, hay que instalar algunas librerías que permitan iniciar el nodo principal en el espacio de trabajo de MATLAB. Una vez hecho esto ya se puede establecer la comunicación con el robot, ver su estado, ver la información de los diferentes nodos presentes en la comunicación…

2.3 Robot Operating Systems (ROS)

ROS es una serie de librerías y herramientas que permiten desarrollar el software de diferentes robots.

Robotics System Toolbox provee algoritmos para la comunicación y el desarrollo de aplicaciones con robots. Esta es una de las herramientas que se utilizan en este proyecto. Con ella se puede diseñar y desarrollar algoritmos para la representación de mapas, planear rutas… Se pueden crear vistas de los diferentes robots y ver su estado, muy útil para la realización de este proyecto.

Esta System Toolbox contiene interfaces entre MATLAB y ROS que es la que hace posible que se pueda realizar la comunicación con el brazo IRB120. Contiene además simulaciones de diferentes robots para poder verificar los resultados de las diversas aplicaciones.

A través de ROS se va a realizar la comunicación entre el brazo robot y la herramienta de trabajo MATLAB. Va a ser la encargada de desarrollar el algoritmo que permite dicha comunicación, la que va a leer la información que llega en los Subscribers o mandar las posiciones al robot escribiendo en los Publishers.

Para poder ver la información que se está mandando o recibiendo del robot hay que prestar atención a los nodos. Cada nodo o topic contiene una información distinta y de tipo distinto. Estos tipos pueden ser de tipo string, de tipo pose… Para poder ver lo que contienen, hay que crearse un Subscriber. Este lo que hace es subscribirse al nodo que se le diga y leer lo que contienen los mensajes del nodo.

Se escribe “rostopic list” para ver todos los nodos activos.

$ rostopic list

Después hay que subscribirse al nodo, en este caso hay que subscribirse en el que se va a trabajar: **"/Object_pose"**.

Este nodo es del tipo **"geometry_msgs/Pose"**, por lo que el Subscriber va a ser del mismo tipo.

$ posesub=rossubscriber('/Object_pose')

Ya está el Subscriber creado. Ahora hay que crear un mensaje que reciba la información del Subscriber.

$ posedata=receive(posesub)

Al realizar lo anterior se obtiene el contenido de ese nodo.

Además de escuchar al nodo y ver su contenido, hay que publicar también la información que se le quiere mandar. Esto se realiza a través de un Publisher:

Se va a influir sobre el nodo anterior.

$ posepub=rospublisher('/Object_pose','geometry_msgs/Pose');

Esta vez no sólo hay que decirle en qué nodo se quiere publicar, sino que también hay que decir qué tipo de mensajes se quiere publicar.

Para poder publicar en ese nodo hay que crear un mensaje del mismo tipo que el Publisher.

$ posemsg=rosmessage(posepub);

Para poder ver el tipo de mensaje que se ha creado, se puede hacer:

$ showdetails(posemsg);

Aparecerá el contenido del mensaje y su estructura.

Dependiendo del tipo de mensaje habrá que escribir el contenido de una forma o de otra que se contará más adelante. Por último, se publica el mensaje en el nodo a través del Publisher.

$ send(posepub,posemsg);

De esta forma, se pueden ver los contenidos de diferentes nodos y actuar de acuerdo a su posición, velocidad… Por ejemplo, para ver dónde se encuentran todas las partes del robot, se puede crear un Subscriber a **"/joint_states"** que devuelve la posición de cada parte.

En cambio para influir sobre esas partes, hay que publicar en **"/joint_path_command"**, el cual es el encargado de realizar los movimientos del brazo IRB120.

Para realizar esta comunicación posible entre MATLAB y ROS, lo que se utiliza es la filosofía del listener y el talker, del cual se hablará más adelante.

2.4 ROS Industrial

ROS industrial es un recurso más del que se dispone al utilizar ROS el cual extiende las numerosas capacidades que tiene a nuevas aplicaciones industriales. Es usado para procesos industriales más complejos.

Gracias a ROS Industrial se realiza la comunicación entre ROS y el brazo robótico. Tiene una gran variedad de paquetes desarrollados para establecer una comunicación con distintos robots. Algunos de esos paquetes son los que se han utilizado en este proyecto.

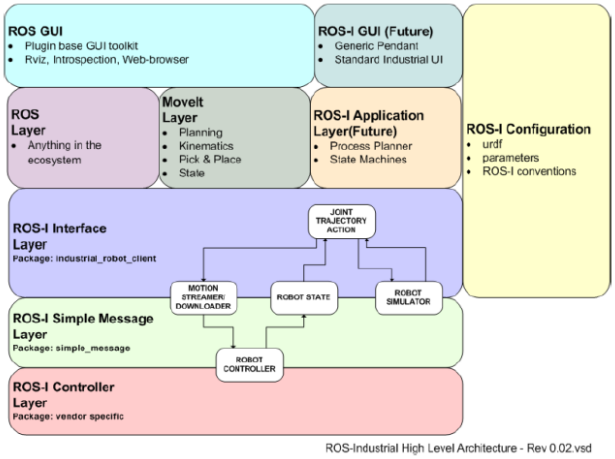


Figura 5: Proceso de comunicación en ROS Industrial.

2.5 RVIZ

ROS incluye algunas herramientas que hacen más fácil el desarrollo de estas tareas. Uno de los más importantes y el que se va a utilizar es RVIZ.

RVIZ (visualización ROS) es un visualizador 3D para la visualización de los datos de sensores y la información del estado de ROS. Usando RVIZ, se puede visualizar la configuración actual del brazo robot ABB en un modelo virtual del robot. También puede mostrar representaciones en vivo de los valores de sensores como los datos de la cámara, las mediciones de distancia de infrarrojos, datos de la sonda, y mucho más.

Para lanzarlo, simplemente hay que escribir en la terminal:

$ rosrun rviz rviz

Muestra una representación tridimensional junto con diferentes componentes para la visualización.

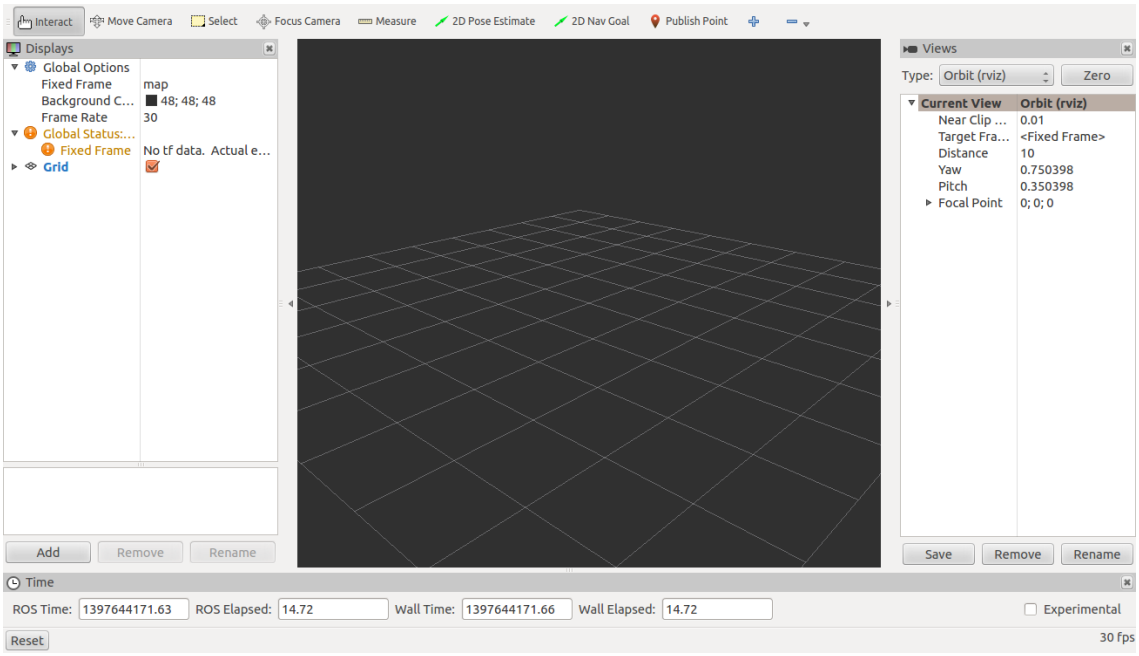


Figura 6: Visor de RVIZ en ROS.

2.6 MoveIt!

Para llevar a cabo la planificación de la trayectoria se utiliza MoveIt!. Esta herramienta incluye las últimas herramientas para planificación, percepción 3D, etc. Es una de las herramientas más utilizadas para simular robots y realizar sus maniobras.

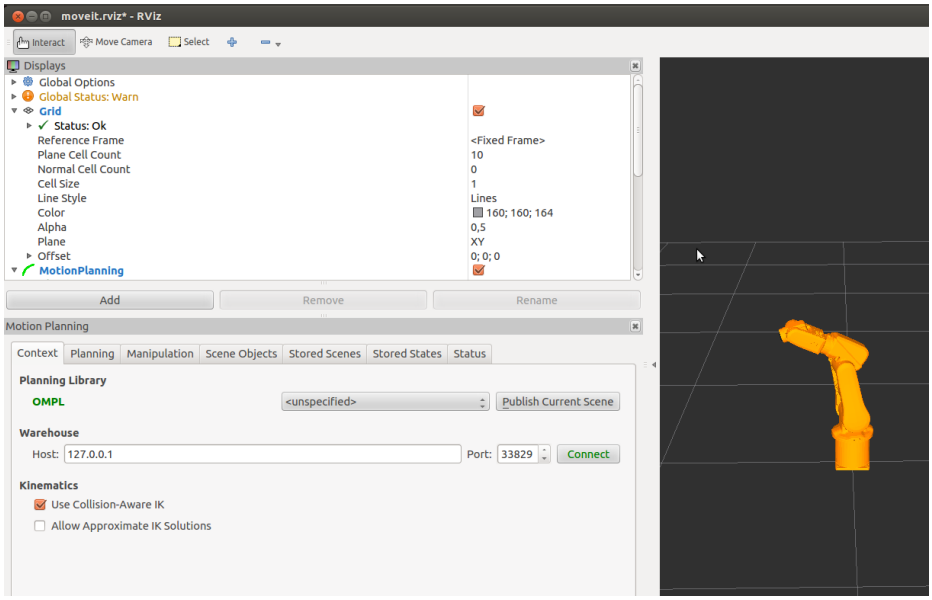


Figura 7: Visor de MoveIt.

Lo primero que se necesita es crear un modelo del robot. Lo que hace MoveIt! es realizar la planificación de las trayectorias permitiendo al usuario interactuar con el robot.

La librería que se necesita para realizar el cálculo de trayectorias es OMPL (Open Motion Planning Library). Esta librería es capaz de realizar algoritmos en diferentes medios (2D y 3D) y en diferentes robots.

2.7 ABB RobotStudio

ABB RobotStudio es un programa desarrollado por ABB para Windows. Es un programa que permite la simulación y creación de software para robots ABB.

RobotStudio proporciona todas las herramientas necesarias para aumentar la rentabilidad del sistema, ya que permite realizar tareas sin afectar a la producción, es decir, enfocado a la programación y optimización offline.

La programación fuera de línea es la mejor manera de maximizar la rentabilidad de inversión de los sistemas de robots. El software de simulación y programación fuera de línea de ABB, RobotStudio, permite efectuar la programación del robot en un ordenador en la oficina sin interrumpir la producción.

Esto proporciona numerosos beneficios incluyendo:

- La reducción de riesgos.
- Inicio más rápido.
- Cambio más corto.
- Productividad incrementada.

RobotStudio se basa en el ABB VirtualController, una copia exacta del software real que ejecuta sus robots en producción. Esto permite realizar simulaciones muy realistas, utilizando programas reales de robot y archivos de configuración idénticos a los utilizados en la planta.

En el caso de este proyecto, el programa RobotStudio va a ser usado para recibir los comandos que recibiría un robot real y realizar su simulación.

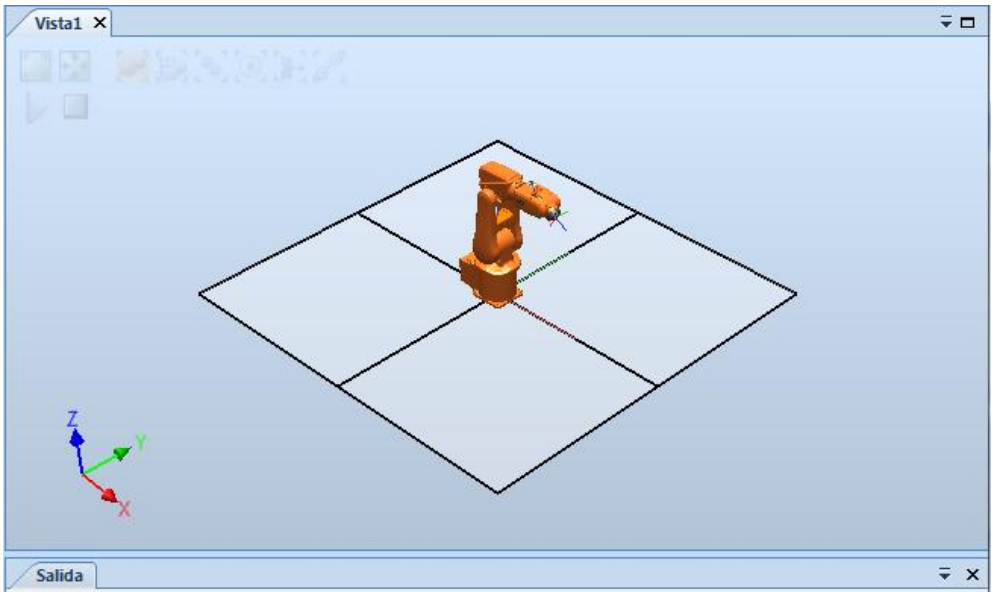


Figura 8: Visor del espacio de trabajo de RobotStudio.

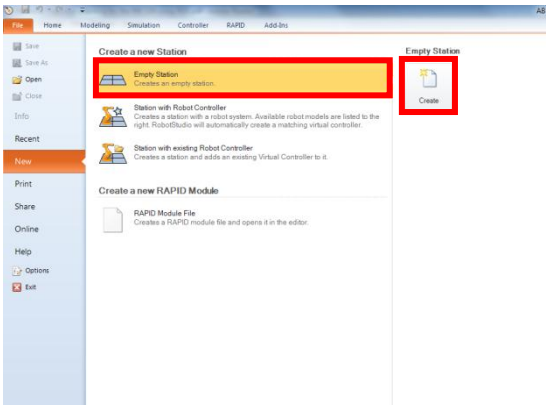
Capítulo 3

Preparación de la simulación en Robotstudio

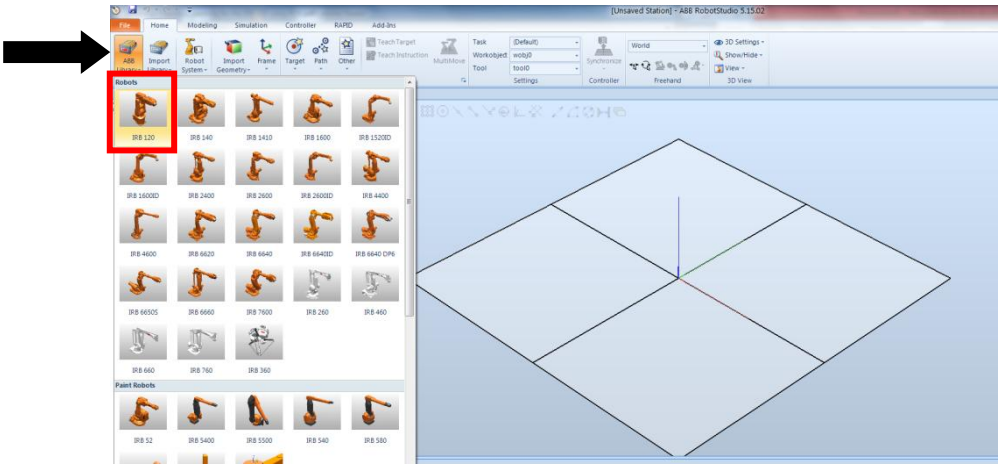
Para poder llevar a cabo la simulación del proyecto, primero hay que crear y configurar una estación de trabajo y un brazo robot, en este caso, el IRB120.

3.1 Inicializar RobotStudio

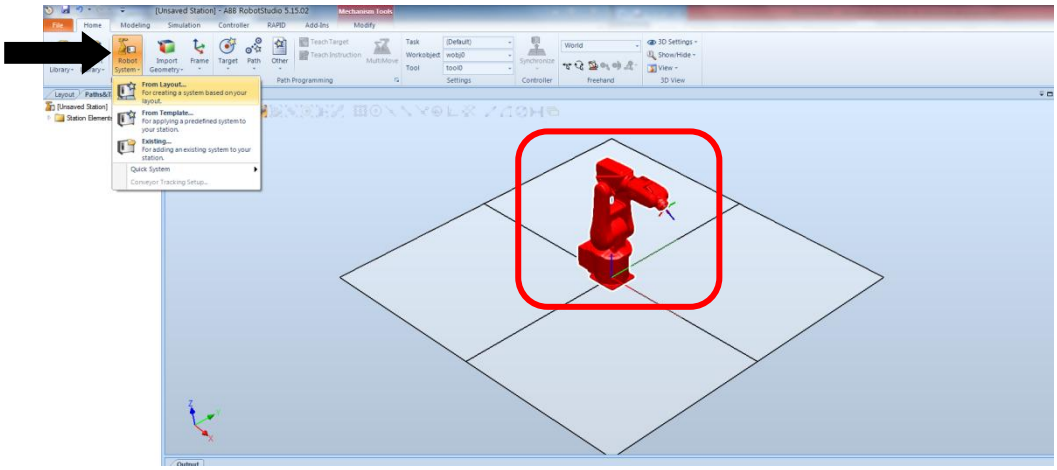
Se inicializa RobotStudio y se selecciona “Empty Station”.



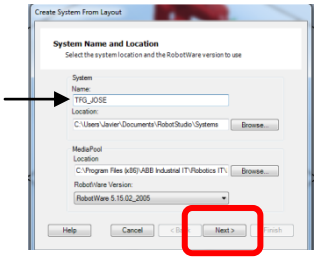
Una vez se ha creado una nueva estación vacía, hay que buscar “ABB Library” y en el desplegable elegir el robot que se va a simular, en caso de este proyecto el IRB120.

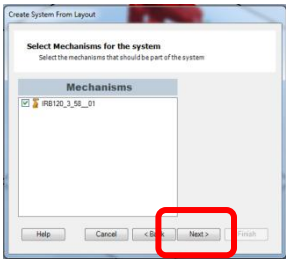


Se selecciona “Robot system” y hay que clicar en “from Layout”.

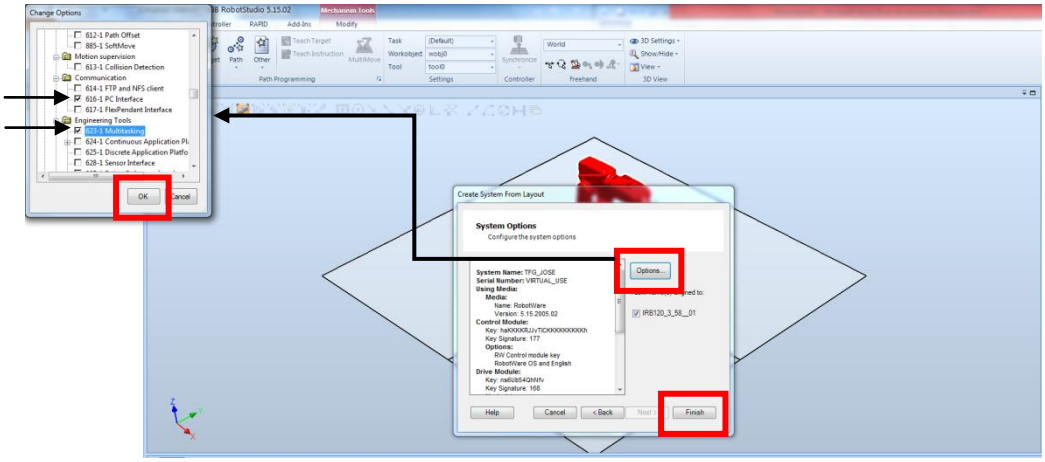


Ahora hay que elegir un nombre para el sistema que se va a crear, una vez elegido hay que pulsar en “next”. Aparece otra ventana, en la cual sólo hay que clicar en “next”.



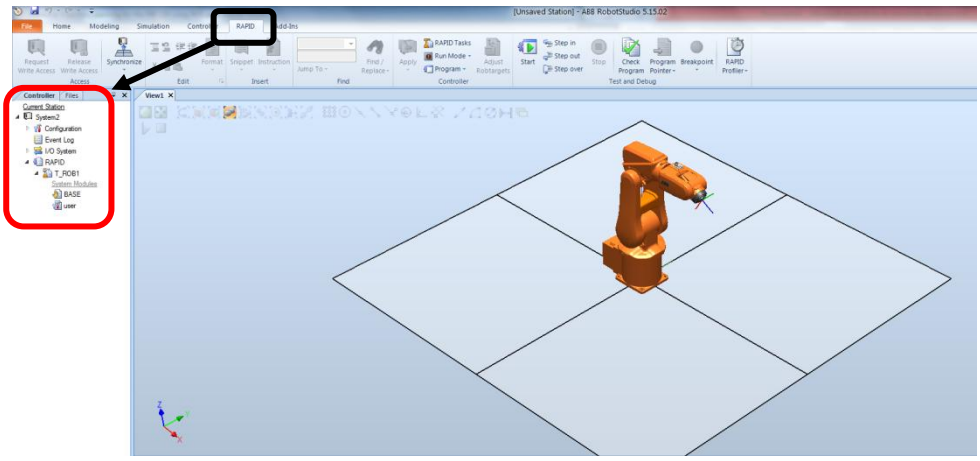


Por último, se llega a una última ventana en la que hay que elegir las opciones del robot. Se clica en “Options…” y en la ventana que aparece hay que buscar “616-1 PC Interface” y “623-1 Multitasking”. Una vez seleccionados en la ventana anterior hay que pulsar en “finish” y se crea el nuevo sistema.



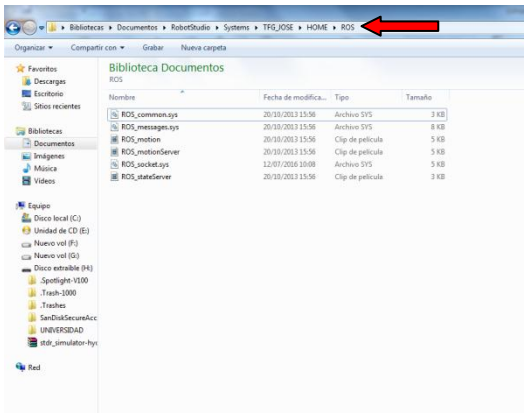
3.2 Asignar archivos RAPID

Hay que fijarse en la parte de RAPID, que es el código con el que se va a programar el robot para que se comunique. Está vacío, por lo que hay que añadir y crear los archivos de RAPID.



Para ello hay que realizar antes unos pasos:

1. Hay que irse a Documentos\RobotStudio\Systems y seleccionar el archivo correspondiente al nombre del sistema que se acaba de crear (TFG_JOSE).
2. Dentro de esta carpeta, hay que clicar en HOME y crear una carpeta dentro de esta que se llame ROS.
3. Se introduce en esta carpeta "ROS" los archivos de configuración del Rapid del robot.

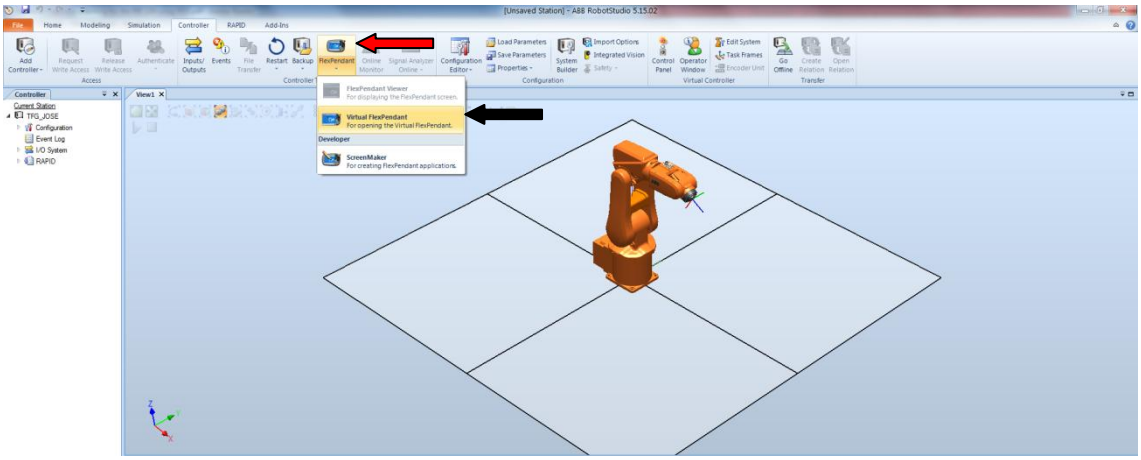


4. Hay que mirar cuál es la IP del ordenador en el que se está simulando (en este caso, el ordenador de Windows) e introducirla en el archivo llamado ROS_socket.sys. Hay que fijarse que hay una línea con "GetSysInfo(\LanIp)". Ahí es donde hay que introducir la IP del ordenador.
5. Si la IP en la que se está conectado no es fija, hay que tener cuidado; ya que puede cambiar y entonces se perdería la comunicación entre el ordenador emisor y el receptor.

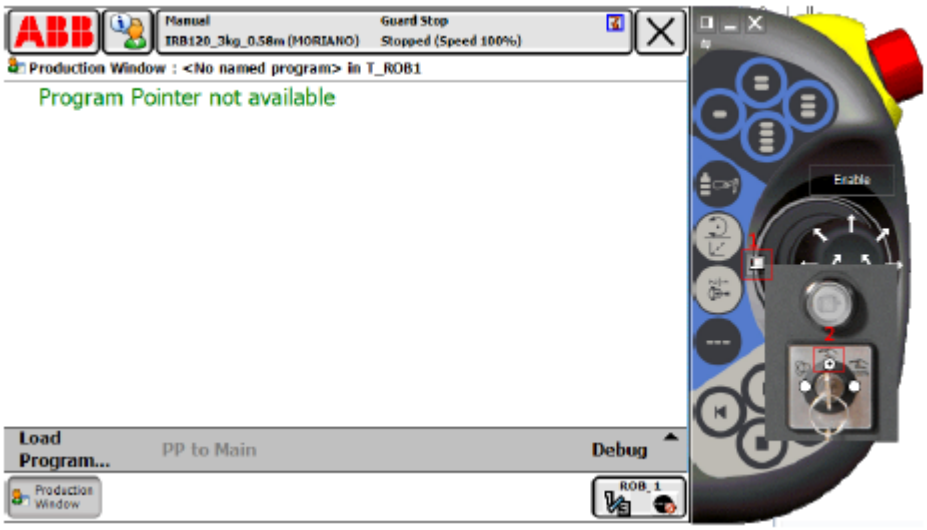
3.3 Virtual Flex Pendant

Para poder interactuar con el IRC5 existe un controlador llamado FlexPendant, y en RobotStudio este controlador se llama Virtual FlexPendant.

Hay que seleccionar este controlador para poder usar el modo manual y crear las tareas y los módulos necesarios. Para seleccionarlo, hay que clicar en la pestaña de "controller", después en "FlexPendant" y en el desplegable que aparece en "Virtual FlexPendant".



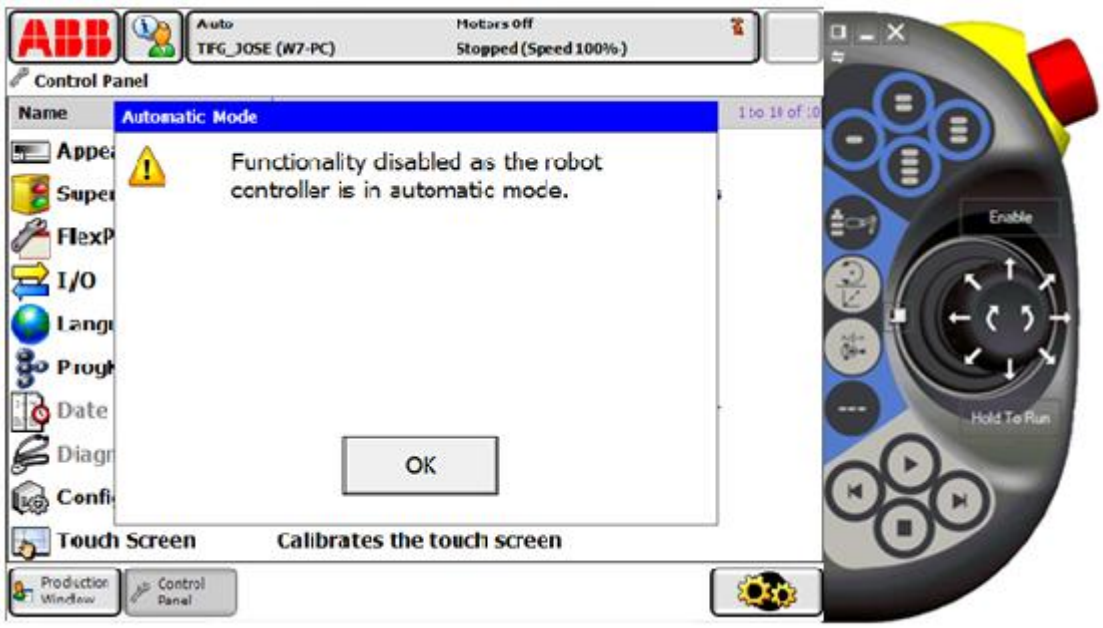
Una vez aparece el controlador virtual, hay que elegir el modo manual. Para ello se selecciona el botón a la izquierda del joystick y se gira la llave a la posición central.



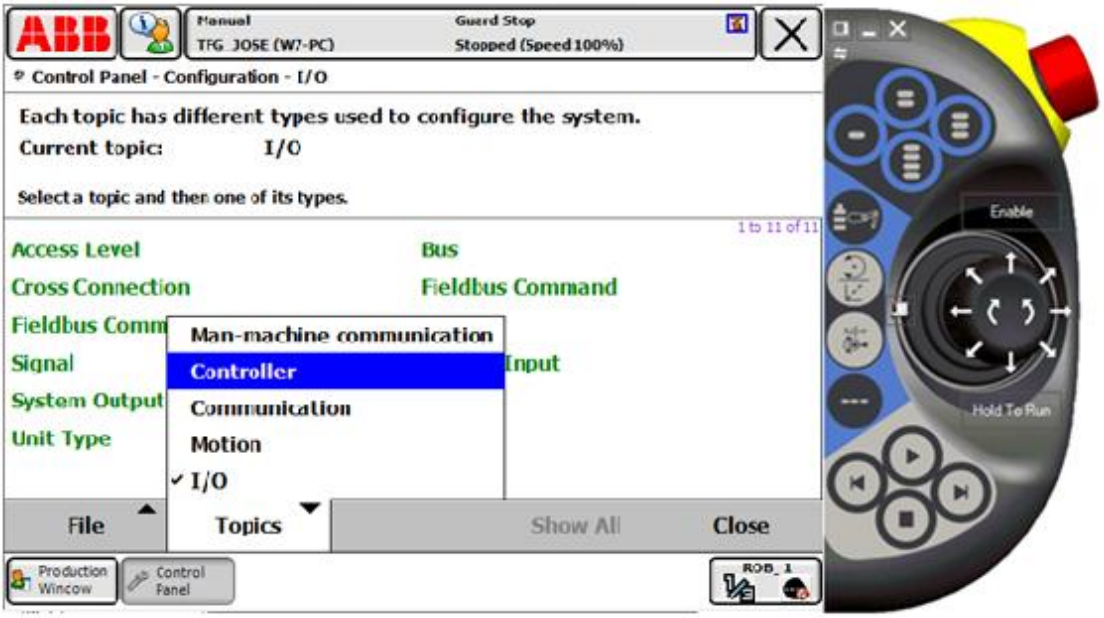
3.4 Instalación del socket

Para poder instalar el socket, hay que ir añadiendo los archivos que se han copiado anteriormente en la carpeta HOME. Para ello, hay que clicar en “ABB” y seleccionar “Control Panel”.

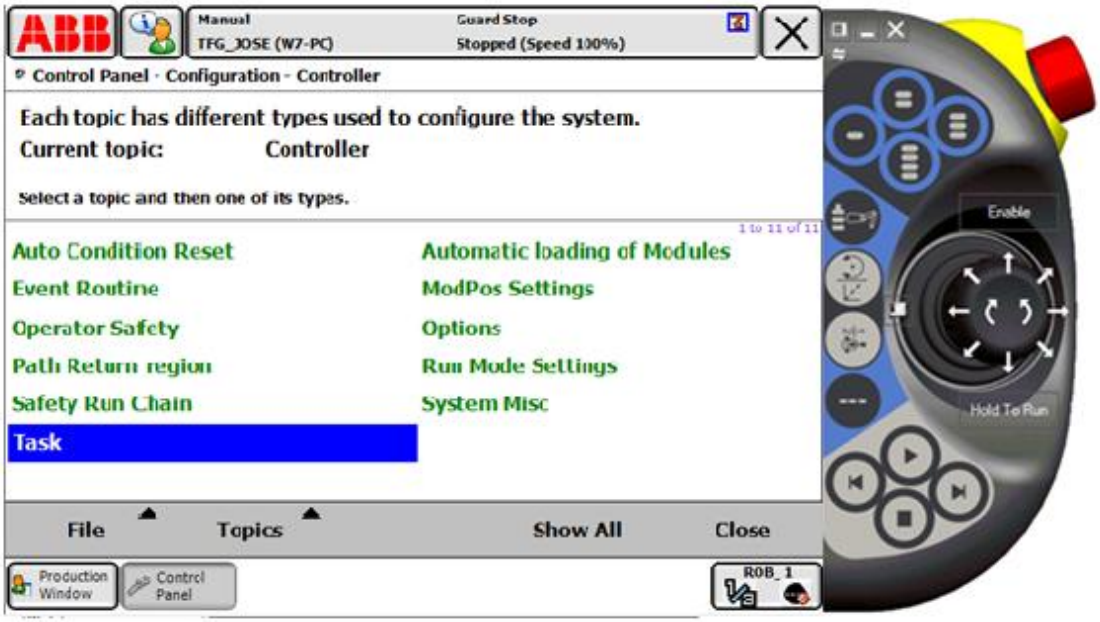
Lo siguiente es seleccionar “Configuration”. En caso de no haber puesto el controlador en modo manual, debería aparecer un mensaje de error como el de la siguiente imagen, por lo que se debería retroceder y ponerlo en modo manual como se ha indicado.



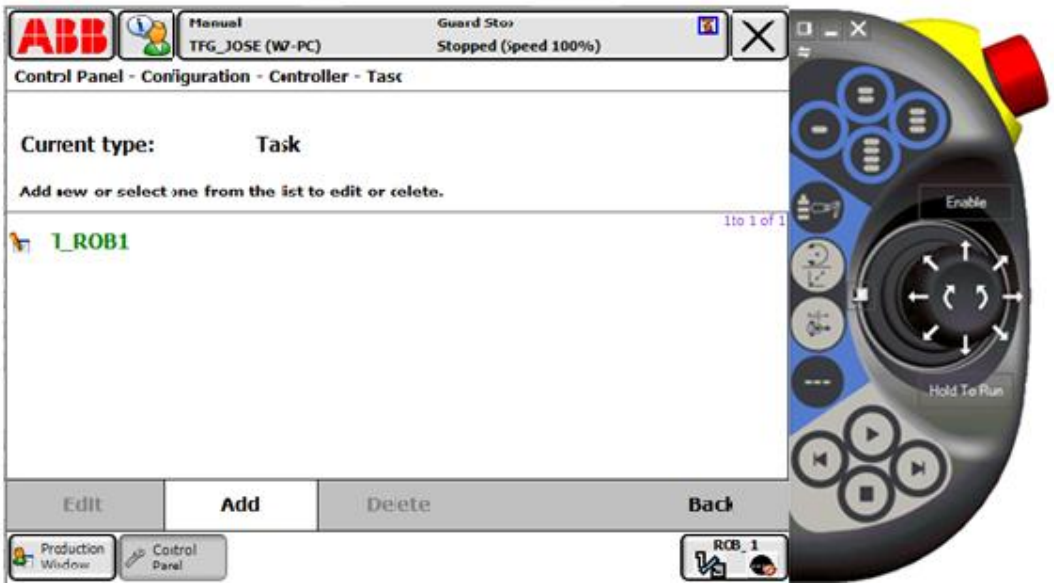
En este nuevo menú, en la parte inferior hay que seleccionar “Topics” y en el desplegable que aparece seleccionar “Controller”.



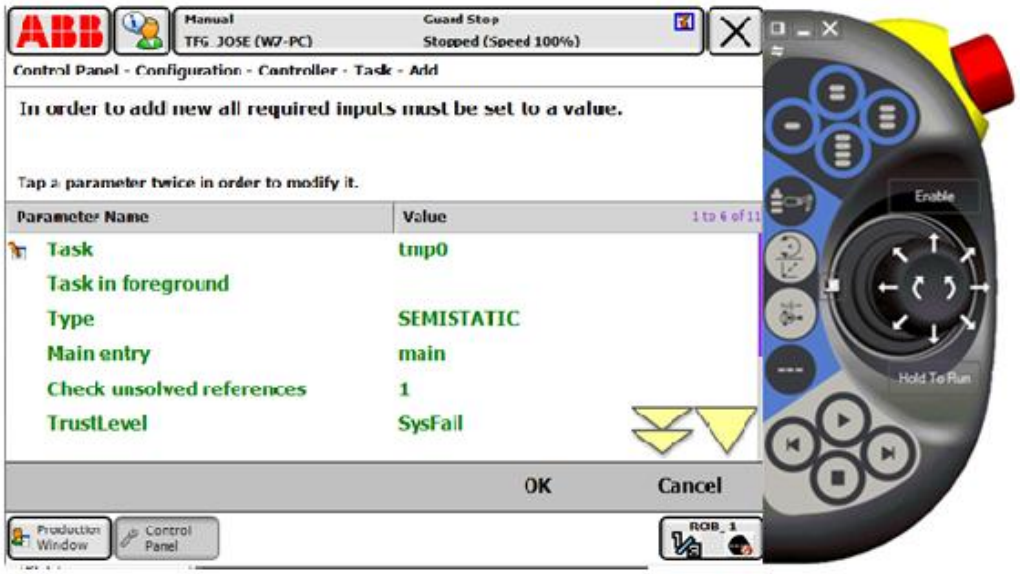
Y por último, para poder ir añadiendo las tareas, hay que hacer doble click en “Task”.



Al hacer esto se puede acceder al menú en el que se encuentran todas las tareas, en el cual sólo se visualiza “T_ROB1”. Hay que pulsar en “Add” para poder ir añadiéndolas:



Una vez en este último menú, hay que ir introduciendo los parámetros y los valores de cada tarea. Al finalizar, pulsar en “OK”.

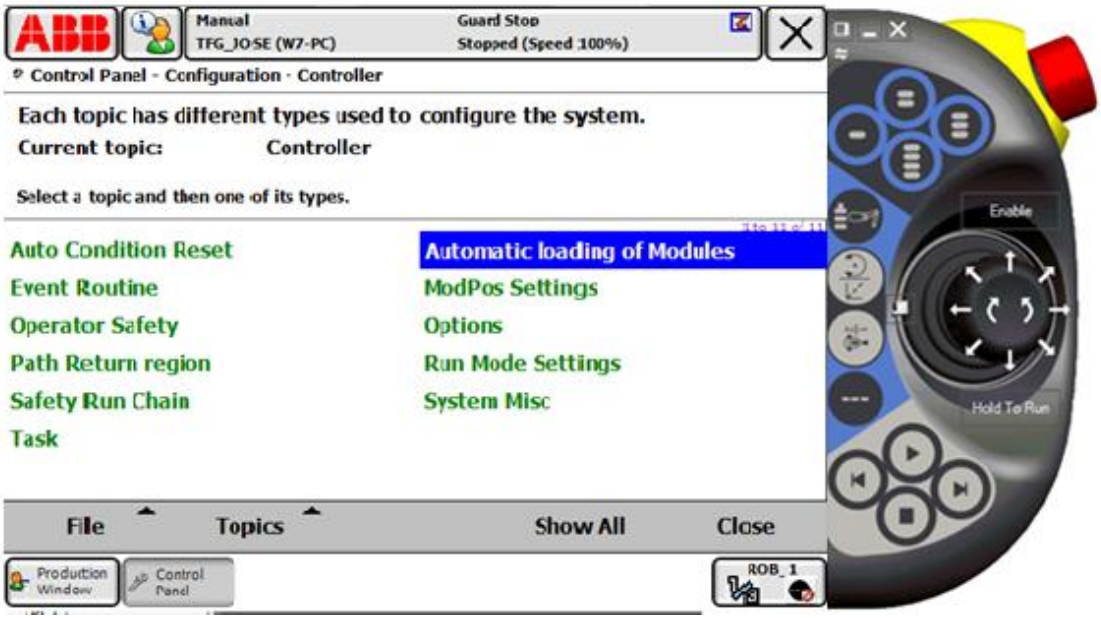


Hay que ir añadiendo una a una con la configuración de la tabla inferior.

Name	Type	Trust Level	Entry	Motion Task
ROS_StateServer	SEMISTATIC	NoSafety	main	NO
ROS_MotionServer	SEMISTATIC	SysStop	main	NO
T_ROB1	NORMAL		main	YES

Tabla 1: Lista de tareas.

Cuando se han añadido las tres tareas, falta por añadir los diferentes módulos. Estos se cargarán automáticamente cada vez que se inicia el controlador. Para llegar hasta el menú correspondiente, hay que llegar al mismo en el que se encuentran las tareas, es decir: ABB-> Control Panel-> Configuration -> Controller y aquí se selecciona de todas las opciones "Automatic loading of Modules".

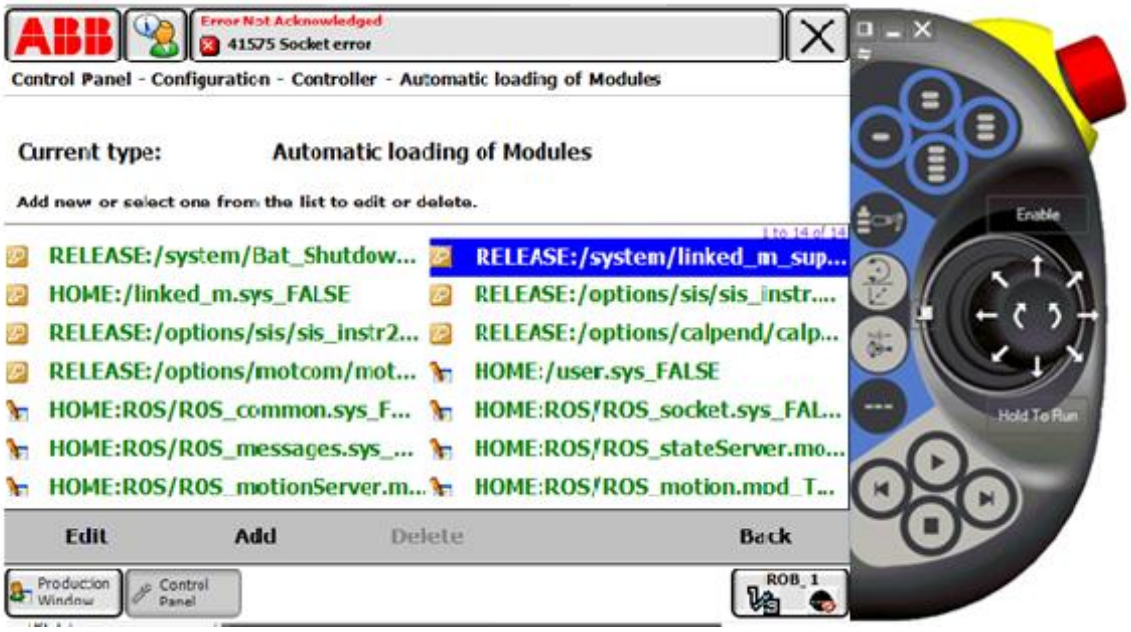


Cuando se accede al menú de todos los módulos hay que dar en "Add" e ir añadiendo y configurando los 6 módulos correspondientes. Su configuración se encuentra en la tabla inferior.

File	Task	Installed	All Tasks	Hidden
HOME:ROS/ROS_common.sys		NO	YES	NO
HOME:ROS/ROS_socket.sys		NO	YES	NO
HOME:ROS/ROS_messages.sys		NO	YES	NO
HOME:ROS/ROS_stateServer.mod	ROS_StateServer	NO	NO	NO
HOME:ROS/ROS_motionServer.mod	ROS_MotionServer	NO	NO	NO
HOME:ROS/ROS_motion.mod	T_ROB1	NO	NO	NO

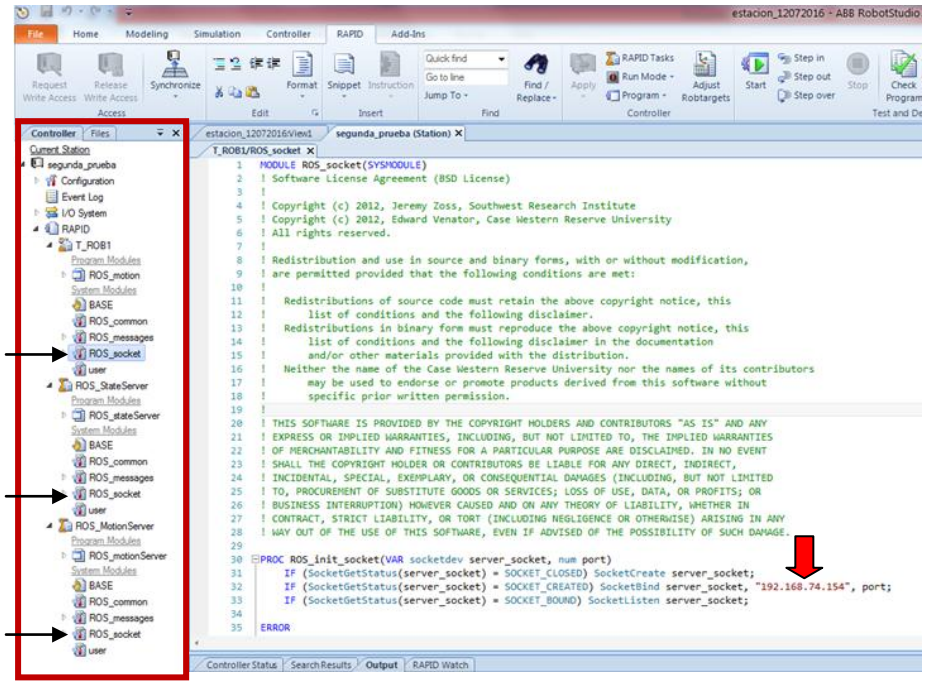
Tabla 2: Lista de módulos.

El aspecto que te queda es el siguiente:

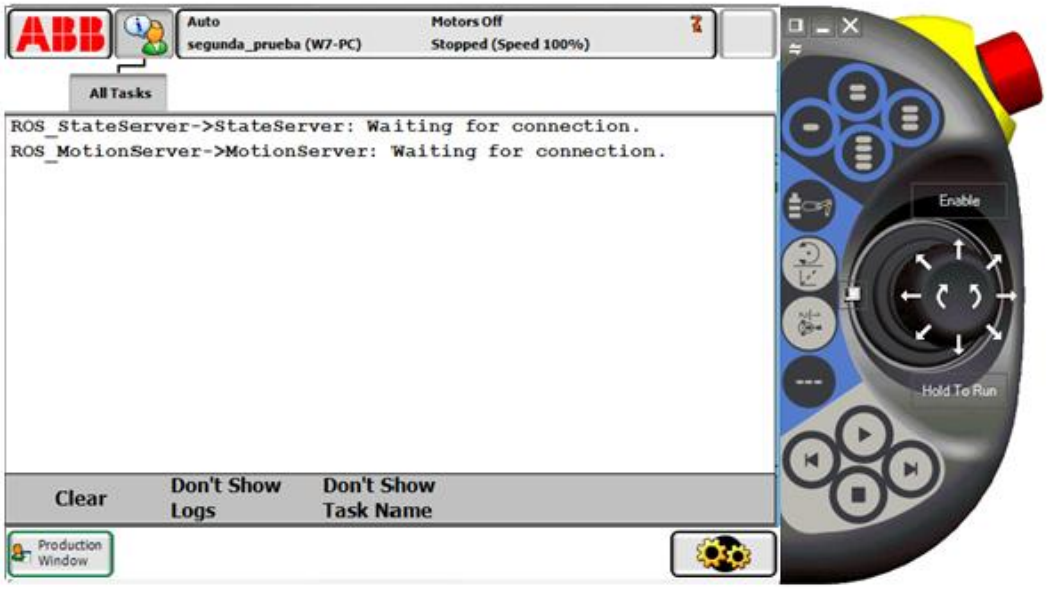


Si todo ha salido bien, ya se ha terminado la configuración del controlador y del robot IRB120. Un mensaje de error muy común es el siguiente:

Si esto ocurre, es porque la dirección IP del ordenador es distinto al que se ha introducido en los archivos RAPID. Para corregir este error, hay que ir a la ventana de RobotStudio "RAPID". En la columna de la izquierda, se puede observar que ahora aparecen los socket que se han introducido. Hay que clicar en RAPID y desplegar los 3 que aparecen. Y dentro de cada uno hay que cambiar en "ROS_socket" la dirección IP y asignarle la dirección correcta.



Una vez se ha solucionado esto, se guarda la configuración de los archivos y se vuelve a abrir el FlexPendant. Ahora tiene que aparecer el siguiente mensaje:



Esto significa que está esperando la conexión del ordenador desde el que se va a controlar el robot (el ordenador con Linux) y RobotStudio estaría ya configurado.

Capítulo 4

ROS

4.1 Instalación paquetes ROS

Para la comunicación con RobotStudio desde el ordenador emisor (ordenador con Linux), se va a necesitar instalar algunos paquetes.

Lo primero de todo es instalar ROS. En el caso de este proyecto, el ROS que se ha instalado es ROS hydro. Dependiendo del tipo de Ubuntu que se tenga hay que instalar uno u otro. Se dispone de Ubuntu 12.04, por lo que en la página oficial de ROS, hay que seguir los pasos para instalar ROS hydro en Ubuntu 12.04.

Cuando se ha instalado ROS, hay que instalar unos cuantos paquetes más, necesarios para la comunicación con RobotStudio.

- ROS Industrial:
  El siguiente paquete se puede instalar directamente si se introduce en un terminal:

  $ sudo apt-get install ros-hydro-industrial-core

- MoveIT:
  Para instalar este paquete hay que hacer lo mismo que el anterior, se escribe en un terminal:

  $ sudo apt-get install ros-hydro-moveit-full

4.2 Conexión con IRB120

Una vez instalados los paquetes anteriores se puede proceder al desarrollo en ROS.

Lo primero que hay que hacer es crearse en la carpeta personal una carpeta en la que se van a guardar todos los archivos y paquetes que contienen el código del proyecto.

Una vez está creada la carpeta, hay que crear otra denominada “packages” y se guardan ahí todos los archivos.

Lo siguiente que se hace es compilar los archivos de esa carpeta de la siguiente forma:

$ cd /…/ packages

$ catkin_make

Cuando acaba la compilación y todo está correcto hay que proceder a conectar con RobotStudio. Para ello hay que establecer la fuente del programa (source). Cada vez que se lance un nuevo programa hay que hacer “source”:

$ source /…/ packages/ devel/ setup.bash

Y una vez se ha realizado lo anterior ya se pueden empezar a lanzar los programas. Para conectar con el simulador en RobotStudio que es la finalidad, hay que escribir el siguiente código:

$ roslaunch abb_common robot_interface robot_ip: xxx.xxx.xxx.xxx

Y escribir la IP correcta del ordenador al que se quiere conectar. Si todo es correcto, en el FlexPendant anteriormente abierto tiene que aparecer el siguiente mensaje:

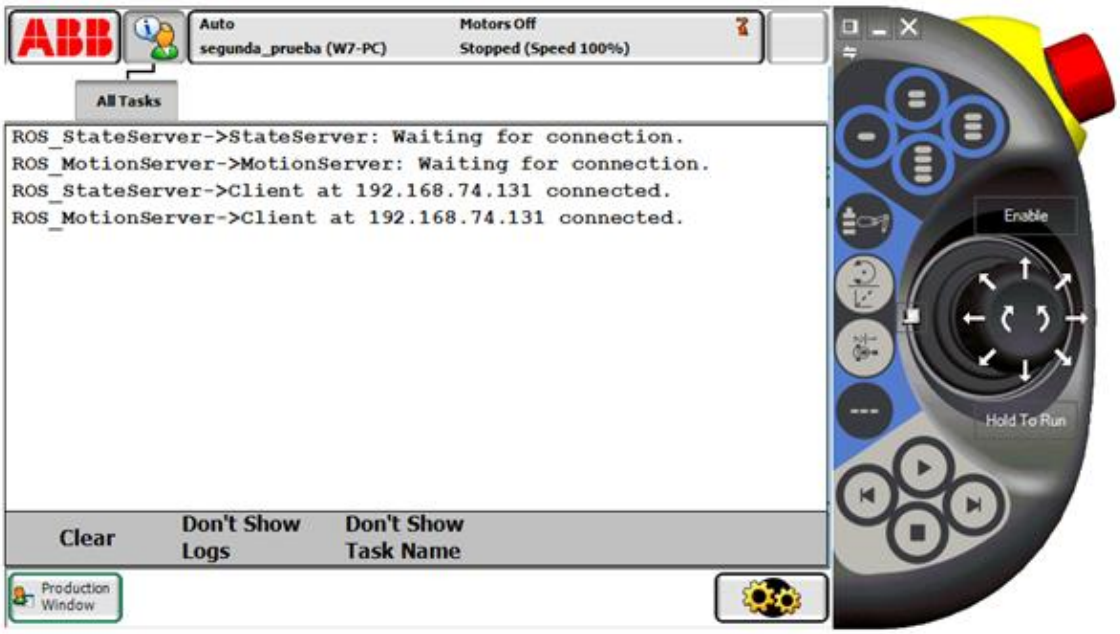


Figura 9: Mensaje de conexión realizada en Virtual FlexPendant.

Una vez que se ha conectado con el robot, hay que abrir un nuevo terminal para investigar los nodos que hay disponibles. Para poder ver todos los nodos que están activos, se puede usar por ejemplo “rostopic list”:

$ rostopic list

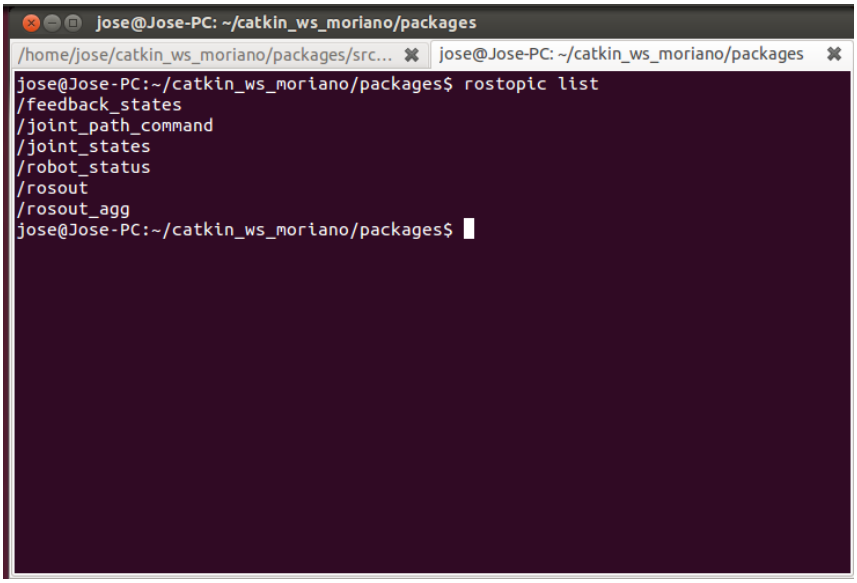


Figura 10: Lista de topics activos.

Uno de los nodos más importantes y que se van a utilizar es **"/joint_states".** Este nodo se encarga de informar sobre la posición de cada parte del robot. Para ver la posición de cada parte hay que introducir en el terminal lo siguiente:

$ rostopic echo /joint_states

Informa a tiempo real de dónde se encuentra cada parte, por lo que si se mueve desde RobotStudio alguna parte del robot, las variables joint_1…joint_6 se actualizarán.

Otro nodo muy importante es **"/joint_path_command"**. Este es el encargado de mandar al robot los puntos a los que se debe mover. Para poder realizar el movimiento, hay que mandarle punto a punto, es decir, una trayectoria de puntos alcanzables. Y para realizar esto se ha utilizado RVIZ.

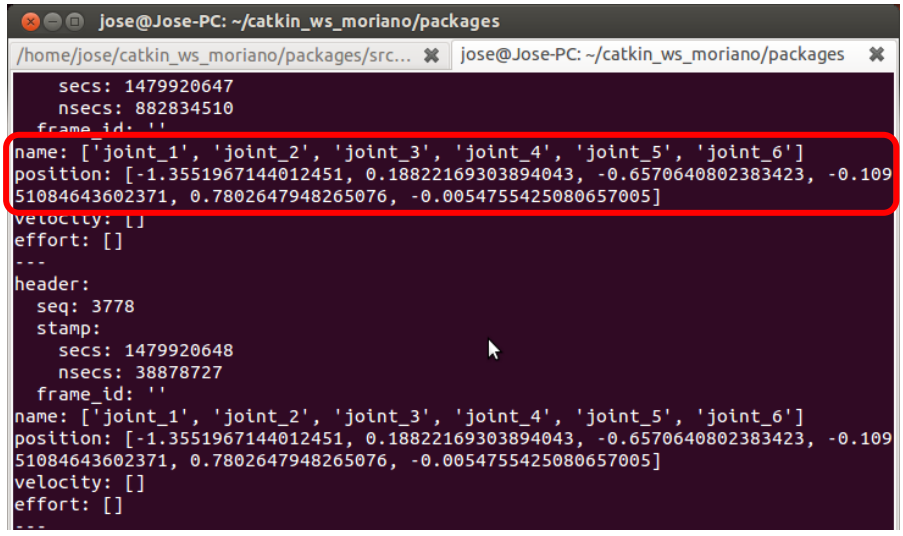


Figura 11: Información recibida en "/joint_states".

4.3 MoveIt!

MoveIt! es el encargado del cálculo de la trayectoria del robot. Para abrirlo, hay que abrir un nuevo terminal e introducir lo siguiente:

$ source /…/ packages/ devel/ setup.bash

$ roslaunch irb120_moveit_config move_group_fast.launch

RVIZ empezará a analizar si el archivo move_group_fast.launch que se encuentra en la misma carpeta que en el caso anterior, no tiene ningún error en la compilación. Tras realizar este análisis, el programa se abrirá y en él se puede ver a la derecha una simulación del robot y a la izquierda se puede ver un menú con todas las opciones que se ofrecen, además de la librería que se va a encargar de llevar a cabo el cálculo de las trayectorias (OMPL).

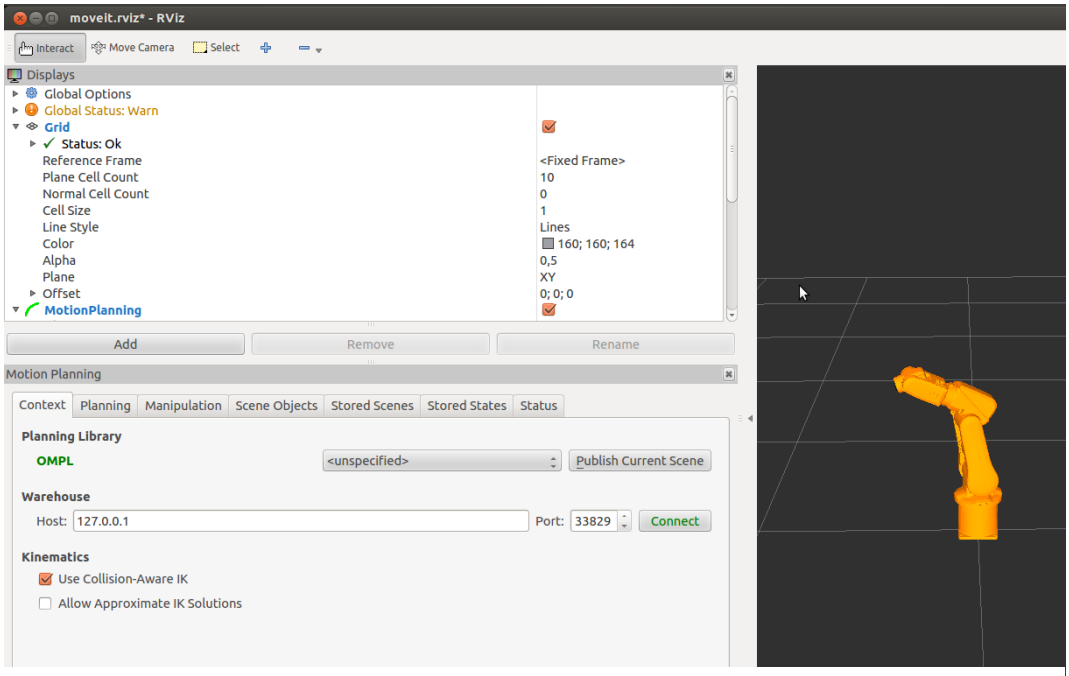


Figura 12: Ventana de RVIZ.

Como el ordenador está conectado con RobotStudio, RVIZ recibe la información del robot a través del nodo **"/joint_states".** Y debido a lo anterior, la simulación del robot (a la derecha) aparecerá en la posición exacta en la que se encuentra en RobotStudio.

Más adelante se le pasará un punto y a través de ese punto, empezará a calcular la trayectoria. En caso de no poder alcanzarla, manda un mensaje de warning y el robot no se moverá. Si se le manda una posición correcta, comunica el número de puntos que ha calculado y la trayectoria que va a seguir.

4.4 Archivo follow

El archivo que tiene todo el código se llama follow.

```
#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit/planning_interface/planning_interface.h>
#include <moveit/robot_trajectory/robot_trajectory.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
#include <moveit_msgs/RobotTrajectory.h>

#include <trajectory_msgs/JointTrajectory.h>
#include <geometry_msgs/Pose.h>

#include <signal.h>
#include <ROS/xmlrpc_manager.h>
```

Lo primero que hay que hacer es incluir y declarar en el archivo fuente todas las librerías .h que se van a necesitar.

```
ROS::Publisher joint_path_command;

bool motion=true;
bool painting=true;
bool working=false;
bool check=false;
bool success=false;
bool finish=false;
geometry_msgs::Pose pose2;
geometry_msgs::Pose pose3;
geometry_msgs::Pose initial;
float position1 = 0.0;
float position2 = 0.0;
float orientation = 0.0;
moveit_msgs::CollisionObject Sphere;

sig_atomic_t volatile g_request_shutdown = 0;
```

En la primera línea de código, se está creando un publicador a **"/joint_path_command"**, debido a que este nodo es el que se encarga de ordenar los movimientos al robot.

Después se declaran las variables del programa fuente.

```
void mySigIntHandler(int sig)
{
  g_request_shutdown = 1;
}

void shutdownCallback(XmlRpc::XmlRpcValue& params, XmlRpc::XmlRpcValue& result)
{
  int num_params = 0;
  if (params.getType() == XmlRpc::XmlRpcValue::TypeArray)
    num_params = params.size();
  if (num_params > 1)
  {
    std::string reason = params[1];
    ROS_WARN("Shutdown request received. Reason: [%s]", reason.c_str());
    g_request_shutdown = 1;
  }

  result = ROS::xmlrpc::responseInt(1, "", 0);
}
```

Las dos funciones anteriores permiten parar la simulación.

```
void checking (sensor_msgs::JointState joints)
{
    if(check==true)
    {

      joints.position.resize(6);

      if((joints.position[0]-position1<0.01) && (joints.position[0]-position1>-
0.01) && (joints.position[1]-position2<0.01) && (joints.position[1]-position2>-
0.01))
      {
        finish=true;
        ROS_INFO("Movimiento terminado");
        check=false;
      }
    }
}
```

La función anterior es la encargada del checking. Esta función analiza el nodo **"/joint_states"** cada vez que se manda una nueva posición.

Si el check es activado compara la posición deseada con la posición real del robot. De hecho no compara todas las partes del robot, ya que con 2 partes es suficiente. Compara con las 2 primeras partes del robot.

Una vez que las posiciones reales y las deseadas son las mismas, lo transmite a través de un mensaje (Movimiento terminado), permite recibir una nueva posición y deja de comparar el nodo **"/joint_states".**

```
void path_calc (geometry_msgs::Pose pose1)
{
    ROS_INFO("Nueva posicion recibida");
    if (working==false)
    {
              working=true;
              pose3.position.x=pose1.position.x;
              pose3.position.y=pose1.position.y;
              pose3.position.z=pose1.position.z;

              pose3.orientation.x=0;
              pose3.orientation.y=1;
              pose3.orientation.z=0;
              pose3.orientation.w=0;
              working=false;
    }
}
```

Cuando se llama a la función void path_calc, lo que se está haciendo es recibir una nueva posición. De hecho esa posición es la que se va a mandar desde MATLAB.

En esta función se crea el mensaje que se va a mandar en el nodo **"/Object_pose".** Este nodo es del tipo **geometry_msgs/Pose** que tiene Position y Orientation. Por lo tanto hay que editar este mensaje respetando el tipo de mensaje.

La primera parte es un array de 3 puntos: "x", "y" y "z". Se escribe en pose3 la información del primer array.

La segunda parte es la relacionada con la orientación. Se trata de un cuaternio formado por "x", "y", "z" y "w". En este proyecto se ha estado trabajando con una orientación fija, pero puede ser también transmitida de la siguiente forma:

pose3.orientation.x=pose1.orientation.x
pose3.orientation.y=pose1.orientation.y
pose3.orientation.z=pose1.orientation.z
pose3.orientation.w=pose1.orientation.w

De esta forma se está permitiendo al emisor que escriba en MATLAB la orientación que le quiere mandar al robot, de la misma forma que la posición.

Lo que se consigue con el working=false es dejar preparada la función para la próxima vez que sea llamada entrar y volver a mandar la nueva posición.

```
int main(int argc, char **argv)
{
  ROS::init(argc, argv, "follow", ROS::init_options::NoSigintHandler);
  ROS::NodeHandle node_handle;
  ROS::AsyncSpinner spinner(1);
  spinner.start();

  joint_path_command =
node_handle.advertise<trajectory_msgs::JointTrajectory>("joint_path_command", 1,
true);

  moveit::planning_interface::MoveGroup group("arm");
  group.setPlannerId("RRTConnectkConfigDefault");

  moveit::planning_interface::MoveGroup::Plan my_plan;

  trajectory_msgs::JointTrajectory traj1;

  int points;

  group.clearPathConstraints();

  ROS::Subscriber sub1 = node_handle.subscribe("Object_pose",1,path_calc);
  ROS::Subscriber sub3 = node_handle.subscribe("/joint_states",1,checking);
```

Se inicializa el main. En él se van iniciando todos los elementos que se van a utilizar: el archivo "**follow"**, el nodo **"/joint_path_command"** y MoveIt! (para que calcule la trayectoria necesaria con el punto final). Hay que subscribirse a los nodos **"/Object_pose"** para mandar la posición y a **"/joint_states"** para recibir la posición de todas las partes del robot.

```
  initial.position.x=0.3;
  initial.position.y=0;
  initial.position.z=0.6;

  initial.orientation.x=0;
  initial.orientation.y=1;
  initial.orientation.z=0;
  initial.orientation.w=0;

  pose2.position.x=0.3;
  pose2.position.y=0;
  pose2.position.z=0.6;

  pose2.orientation.x=0;
  pose2.orientation.y=1;
  pose2.orientation.z=0;
  pose2.orientation.w=0;
```

Cuando se inicializa por primera vez el archivo follow, el robot se mueve a una posición inicial. Para ello lo primero que hay que hacer es pasarle esa posición inicial.

```
  group.setPoseTarget(initial);
  success=group.plan(my_plan);

  traj1.header=my_plan.trajectory_.joint_trajectory.header;
  traj1.joint_names=my_plan.trajectory_.joint_trajectory.joint_names;
  traj1.points.resize(1);

 if(success==1)
 {
       ROS_INFO("Yendo a posicion inicial");
       points=my_plan.trajectory_.joint_trajectory.points.size();
       traj1.points[0].positions=my_plan.trajectory_.joint_trajectory.points[poin
ts-1].positions;
       joint_path_command.publish(traj1);
       position1=traj1.points[0].positions[0];
       position2=traj1.points[0].positions[1];
       check=true;
       while((!g_request_shutdown) & (!finish));
           sleep(0.1);
       finish=false;
       success=0;
 }
```

Como se puede observar, se está esperando a recibir la posición. Cuando se ha recibido la posición, success=1, por lo que entra a la siguiente parte. Se recibe el mensaje ("Yendo a la posición inicial"), lo introduce en points y lo que hace es mirar el último punto del array, es decir, le pasa un array de puntos pero el que interesa es el último que es el punto final.

Se guarda en position1 y en position2 las posiciones de 2 partes del robot para poder compararlas y saber cuándo se ha llegado a la posición final llamando a la función "check".

En caso de error el robot se parará, en cambio si se ha terminado el movimiento, se le comunica a través del finish y se cambia success a cero para la siguiente vez que llegue una posición.

```
jose@Jose-PC:~/catkin_ws_moriano/packages$ rosrun follow follow
[ INFO] [1479923331.872078252]: Loading robot model 'irb120'...
[ INFO] [1479923331.872278487]: No root joint specified. Assuming fixed joint
[ INFO] [1479923332.424662418]: joint_1 -2.88 2.88 1
[ INFO] [1479923332.424948558]: joint_2 -1.92 1.92 1
[ INFO] [1479923332.424994024]: joint_3 -1.92 1.22 1
[ INFO] [1479923332.425036837]: joint_4 -2.79 2.79 1
[ INFO] [1479923332.425086145]: joint_5 -2.09 2.09 1
[ INFO] [1479923332.425124628]: joint_6 -6.98 6.98 1
[ INFO] [1479923333.222138482]: Ready to take MoveGroup commands for group arm.
[ INFO] [1479923333.324046482]: Yendo a posicion inicial
[ INFO] [1479923363.921781292]: Movimiento terminado
```

Figura 13: Inicialización del archivo follow.

```
  while(!g_request_shutdown)
  {
      if(pose3.position.z>0.0)
      {
              group.setPoseTarget(pose3);
              if ((pose2.position.x-pose3.position.x)>(0.3) | (pose2.position.x-
pose3.position.x)<(-0.3) | (pose2.position.y-pose3.position.y)<(-0.3) |
(pose2.position.y-pose3.position.y)>(0.3
              {
                     pose2=pose3;
                     success=group.plan(my_plan);
                     ROS_INFO("Planeando trayectoria: %d",points);
                     if(success==1)
                     {
                           ROS_INFO("Trayectoria encontrada");

       points=my_plan.trajectory_.joint_trajectory.points.size();

       traj1.points[0].positions=my_plan.trajectory_.joint_trajectory.points[poin
ts-1].positions;
                           joint_path_command.publish(traj1);
                           position1=traj1.points[0].positions[0];
                           position2=traj1.points[0].positions[1];
                           check=true;
                           while((!g_request_shutdown) & (!finish));
                               sleep(0.1);
                           finish=false;
                           success=0;
                           pose3.position.z=0.0;
                     }
              }
```

La siguiente parte del archivo follow se compone de varias partes.

Lo primero que se hace es analizar la posición “Z”, ya que si es inferior a 0 el robot no va a poder llegar y va a dar un error. En caso de ser superior a 0 se entra en el “if”.

Después hay que analizar las posiciones pose2 y pose3. Lo que se quiere es que no sean los mismos para poder ir a una nueva posición, para ello hay que ir analizando su posición para que cuando sea muy cercana vaya a la siguiente parte del if.

Se calcula la trayectoria que tiene que seguir el robot, se informa de que se ha encontrado la trayectoria y la realiza en caso de que sea posible. Se puede decir que la filosofía es la misma que la posición inicial, se le manda el punto final, se le manda la posición de las 2 primeras partes del robot y se realiza el movimiento.

Normalmente el final de esta parte no la realiza debido a que lo que está haciendo realmente es esperar a que el robot se aproxime a la posición inicial para poder saltar a la siguiente parte del programa.

```
      else if (pose2.position.x!=pose3.position.x
|pose2.position.y!=pose3.position.y)
            {
                  pose2=pose3;
                  success=group.plan(my_plan);
                  ROS_INFO("PLaneando trayectoria: %d",points);
                  if(success==1)
                  {
                        ROS_INFO("Trayectoria encontrada");

      points=my_plan.trajectory_.joint_trajectory.points.size();

      traj1.points[0].positions=my_plan.trajectory_.joint_trajectory.points[poin
ts-1].positions;
                        joint_path_command.publish(traj1);
                        position1=traj1.points[0].positions[0];
                        position2=traj1.points[0].positions[1];
                        check=true;
                        while((!g_request_shutdown) & (!finish));
                            sleep(0.1);
                        finish=false;
                        success=0;
                        pose3.position.z=0.0;
                  }
            }
      }
```

Esta parte es la que continúa a la anterior. Cuando ya está muy cerca, analiza que lo poco que falta sea distinto. Realiza lo que queda de trayectoria y llega hasta el final.

Cuando termina de realizar el movimiento, se manda un mensaje de movimiento terminado y se fuerza la position.z a cero para salir del "if" y poder mandar una nueva posición.

```
      else if(pose3.position.z<0)
         ROS_INFO("Movimiento por debajo del plano");
      else
      {
         ROS_INFO("Esperando nueva posicion");
        sleep(1);
      }
  }
  ROS::spin();
  ROS::shutdown();
  return 0;
}
```

Cada vez que se termina de realizar la trayectoria, se comunica que ha terminado y que está esperando a que se le mande la nueva posición.

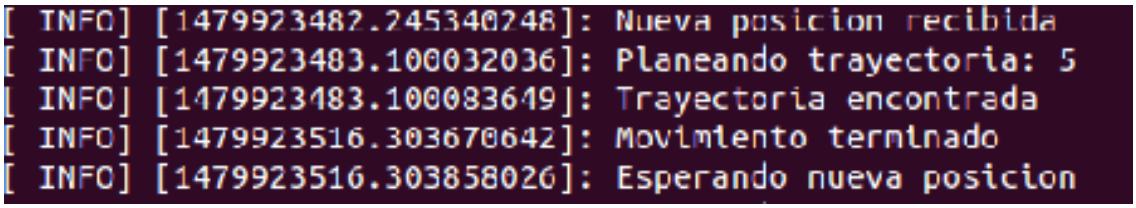


Figura 14: Recepción y trayectoria de la nueva posición.

Para lanzar y simular este archivo, hay que introducir en el terminal:

$ source /…/ packages/ devel/ setup.bash

$ roslaunch follow follow

Capítulo 5

MATLAB

Robotics System Toolbox proporciona una interfaz entre MATLAB y Simulink y el sistema operativo del robot (ROS) que le permite comunicarse con una red de ROS, de forma interactiva explorar las capacidades del robot, y visualizar los datos del sensor. Puede desarrollar, probar y verificar sus algoritmos de robótica y aplicaciones en robots con capacidad de ROS y simuladores de robots como Gazebo y V-REP.

También puede crear una red auto-contenida ROS directamente en MATLAB y Simulink, e importar archivos de registro (ROS rosbags) para visualizar, analizar y post-proceso de los datos registrados. Estas características le permiten desarrollar sus algoritmos de robótica en MATLAB y Simulink, mientras que le da la capacidad de intercambiar mensajes con otros nodos de la red de ROS.

Las principales características le permiten:

- Comunicarse con una red ROS, de forma interactiva explorar las capacidades del robot y visualizar los datos del sensor.
- Crear nodos ROS, editores y suscriptores directamente desde MATLAB y Simulink.
- Crear y enviar mensajes de ROS desde MATLAB y Simulink.
- Crear y enviar mensajes personalizados ROS desde MATLAB y Simulink.
- Utilizar la funcionalidad de ROS en cualquier plataforma (Windows , Linux, Mac).
- Utilizar MATLAB como un maestro de ROS.
- Probar y verificar las aplicaciones en robots habilitados para ROS y simuladores de robots tales como Gazebo y V-REP.
- Crear modelos de Simulink que trabajan con una red de ROS.

La última parte del Proyecto tiene que ver con MATLAB. Para su realización es necesario instalar MATLAB en el ordenador emisor en Ubuntu.

Para inicializar ROS y poder trabajar desde MATLAB, en el workspace hay que escribir:

$ rosinit

Ya se ha iniciado ROS en MATLAB y se pueden ver todos los nodos disponibles.

El nodo que interesa es **"/Object_pose"**. Este es el nodo que se ha creado en el archivo para poder publicar en **"/joint_path_command"**. En realidad, la filosofía que se ha seguido es la de "listener and talker". El programa "follow" lo que está haciendo es esperar a que el "talker" que es MATLAB le mande una posición para poder publicarla en el nodo y realizar el movimiento del robot, que está "escuchando" a MATLAB.

Por lo tanto, lo que se quiere hacer es publicar en **"/Object_pose"**. Para ello, primero se escribe lo siguiente:

$ rostopic info '/Object_pose'

Con el comando anterior se puede ver el tipo del nodo. La información que se tiene de este topic es que es del tipo **geometry_msgs/Pose**, que no tiene ningún Publisher y un solo Subscriber.

Por lo tanto es necesario crear un Publisher asociado a ese nodo para poder escribir un mensaje y publicarlo en él.

$ posepub=rospublisher('/Object_pose','geometry_msgs/Pose')

La estructura anterior lo que viene a decir es que posepub va a ser un Publisher del nodo **"/Object_pose"** del tipo **geometry_msgs/Pose**.

Ahora lo siguiente es crear el mensaje que se quiere mandar del mismo tipo que el Publisher.

$ posemsg=rosmessage(posepub)

Al crear el mensaje se puede ver que es del tipo **geometry_msgs/Pose** y que tiene posición y orientación. Este es el punto que se va a mandar, por lo que se tiene que modificar la posición y la orientación cada vez que se quiera mandar una nueva posición. La forma de ir cambiando la posición que se manda es la siguiente:

$ posemsg.Position.X=0.4

$ posemsg.Position.Y=-0.3

$ posemsg.Position.Z=0.05

$ posemsg.Orientation.Y=1

Si se manda lo anterior por ejemplo y se introduce showdetails(posemsg), devolverá el contenido del mensaje.

Aparecería lo siguiente:

$ showdetails(posemsg)

Position:

X: 0.4

Y: -0.3

Z: 0.05

Orientation

X: 0

Y: 1

Z: 0

W: 0

Y por último, lo que queda es mandar la nueva posición:

$ send(posepub,posemsg)

Lo que quiere decir la instrucción anterior es que se va a mandar el mensaje "posemsg" a "posepub" que al ser a la vez un Publisher de **"/Object_pose"**, publicará su contenido en dicho nodo, que será escuchado por el listener "follow" y publicará en **"/joint_path_command"** una trayectoria calculada por Moveit. Y todo lo anterior junto provocará que el robot de RobotStudio realice un movimiento hasta el punto indicado.

Esto se puede mandar sentencia a sentencia o preparase un archivo .m para mandar todo junto de la siguiente forma:

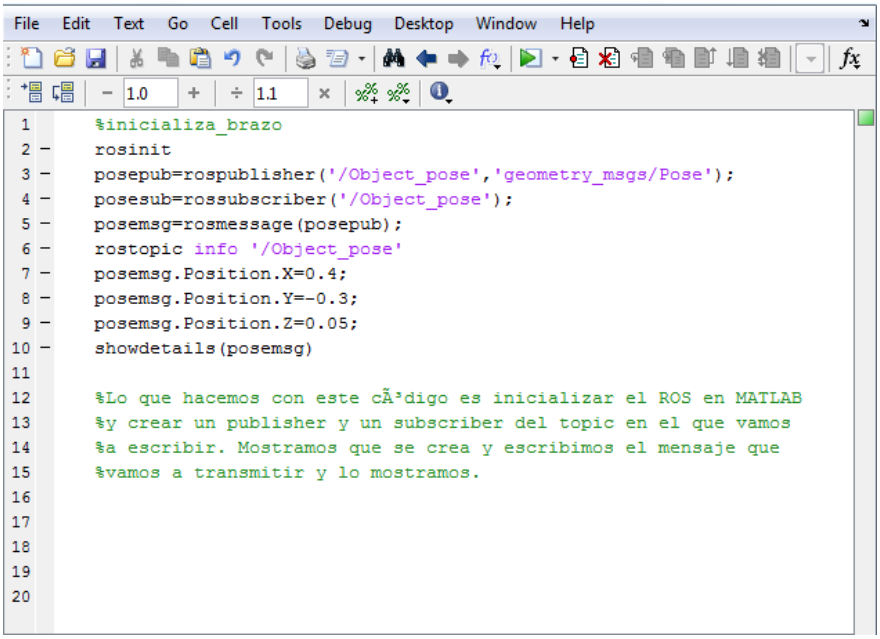


Figura 15: Archivo nueva_posicion.m.

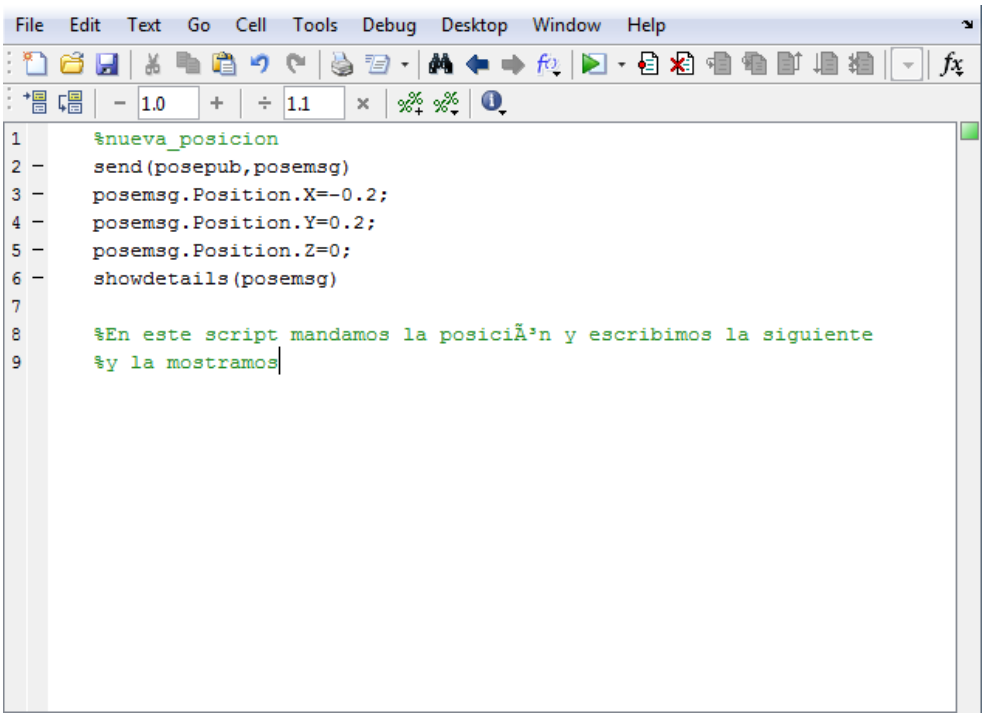


Figura 16: Archivo inicializa_brazo.m.

Capítulo 6

Resultados

La aplicación se ha llevado a cabo en un ordenador con Windows y RobotStudio y otro ordenador con Ubuntu y MATLAB.

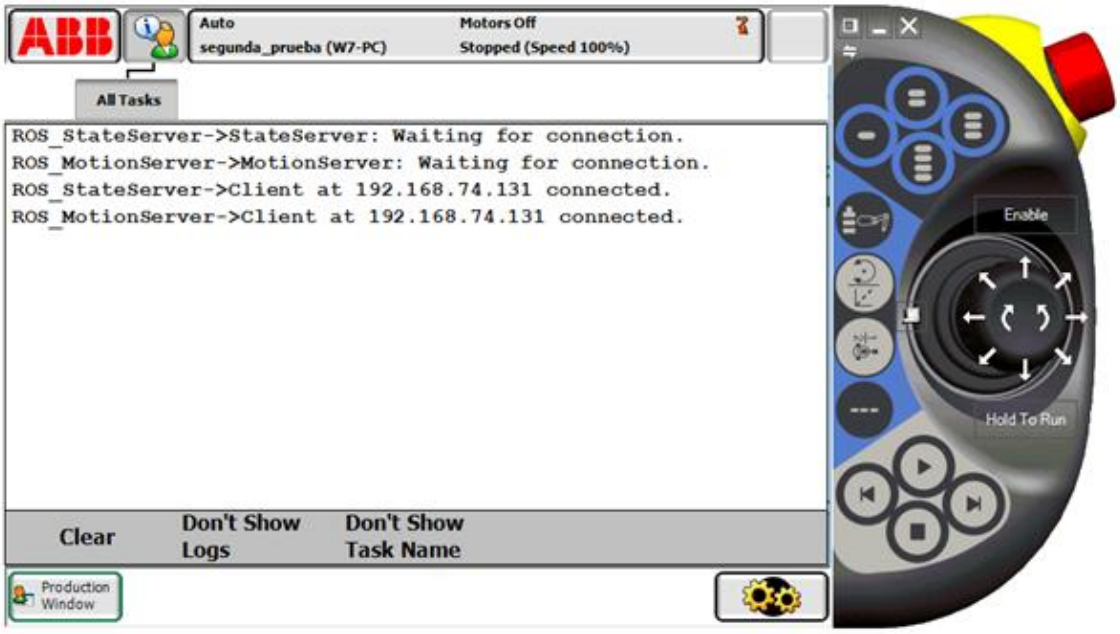


Figura 17: Conexión establecida.

En el ordenador con RobotStudio (el ordenador con Windows) se crea un robot y se conecta con ayuda del Virtual FlexPendant al ordenador con MATLAB (el ordenador con Linux).

Para que vaya a la posición inicial, se desplaza el robot a una posición cualquiera antes de iniciar el archivo follow.

Después se ejecuta MoveIt y el archivo follow para el cálculo de trayectorias e inicializar los nodos necesarios.

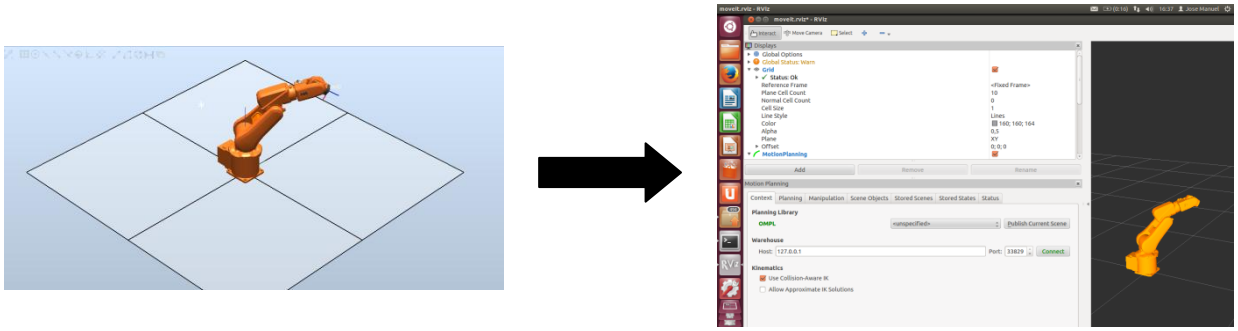


Figura 18: RobotStudio y MoveIt en la misma posición.

Por último se crea en MATLAB la nueva posición del robot y se manda para mover en RobotStudio el robot simulado que está a la espera en la posición inicial.

```
jose@Jose-PC:~/catkin_ws_moriano/packages$ rosrun follow follow
[ INFO] [1479923331.872078252]: Loading robot model 'irb120'...
[ INFO] [1479923331.872278487]: No root joint specified. Assuming fixed joint
[ INFO] [1479923332.424662418]: joint_1 -2.88 2.88 1
[ INFO] [1479923332.424948558]: joint_2 -1.92 1.92 1
[ INFO] [1479923332.424994024]: joint_3 -1.92 1.22 1
[ INFO] [1479923332.425036837]: joint_4 -2.79 2.79 1
[ INFO] [1479923332.425086145]: joint_5 -2.09 2.09 1
[ INFO] [1479923332.425124628]: joint_6 -6.98 6.98 1
[ INFO] [1479923333.222138482]: Ready to take MoveGroup commands for group arm.
[ INFO] [1479923333.324046482]: Yendo a posicion inicial
[ INFO] [1479923363.921781292]: Movimiento terminado
[ INFO] [1479923363.922075114]: Esperando nueva posicion
[ INFO] [1479923364.922405951]: Esperando nueva posicion
[ INFO] [1479923365.922704873]: Esperando nueva posicion
[ INFO] [1479923366.923187512]: Esperando nueva posicion
[ INFO] [1479923367.923773307]: Esperando nueva posicion
[ INFO] [1479923368.924151219]: Esperando nueva posicion
[ INFO] [1479923369.924455860]: Esperando nueva posicion
[ INFO] [1479923370.924685086]: Esperando nueva posicion
```

Figura 19: Se lanza el archivo follow.

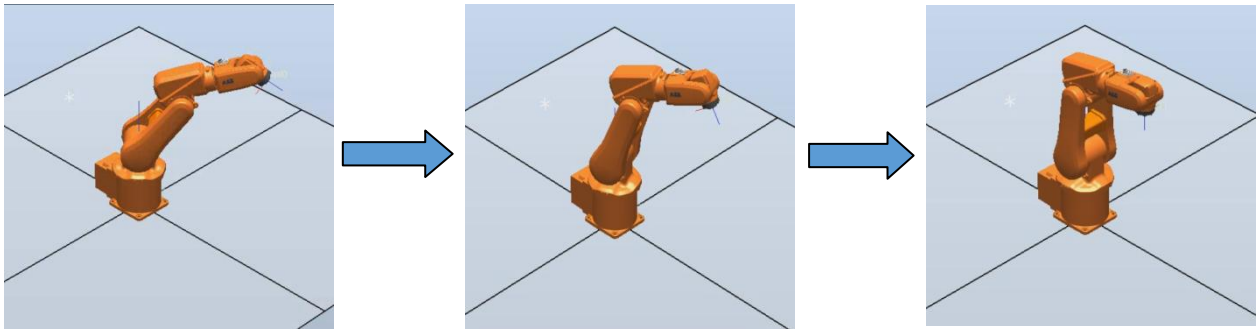


Figura 20:Robot va hacia la posición inicial.

```
[ INFO] [1479923481.958062373]: Esperando nueva posicion
[ INFO] [1479923482.245340248]: Nueva posicion recibida
[ INFO] [1479923483.100032036]: Planeando trayectoria: 5
[ INFO] [1479923483.100083649]: Trayectoria encontrada
[ INFO] [1479923516.303670642]: Movimiento terminado
[ INFO] [1479923516.303858026]: Esperando nueva posicion
[ INFO] [1479923517.304162742]: Esperando nueva posicion
```

Figura 21: Se lanza el archivo nueva_posicion.m.

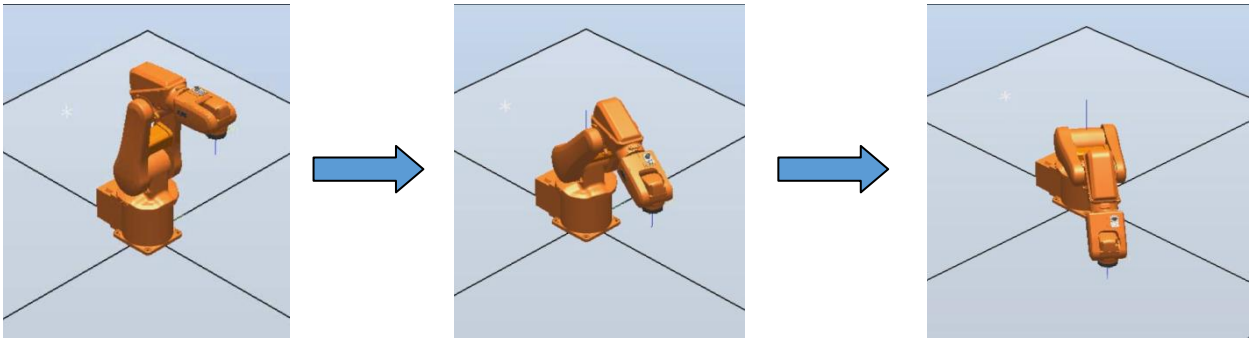


Figura 22: El robot va hacia la posición de destino.

Capítulo 7

Conclusiones y futuro trabajo

7.1 Conclusiones

En este proyecto se ha podido implementar MATLAB y el brazo robot IRB120 (simulado desde RobotStudio).

Ha sido posible controlar el brazo robot desde MATLAB mandándolo a distintas posiciones, tanto a una posición inicial como final. Además del control del brazo, también se ha podido conocer la posición de cada parte del robot.

7.2 Futuro trabajo

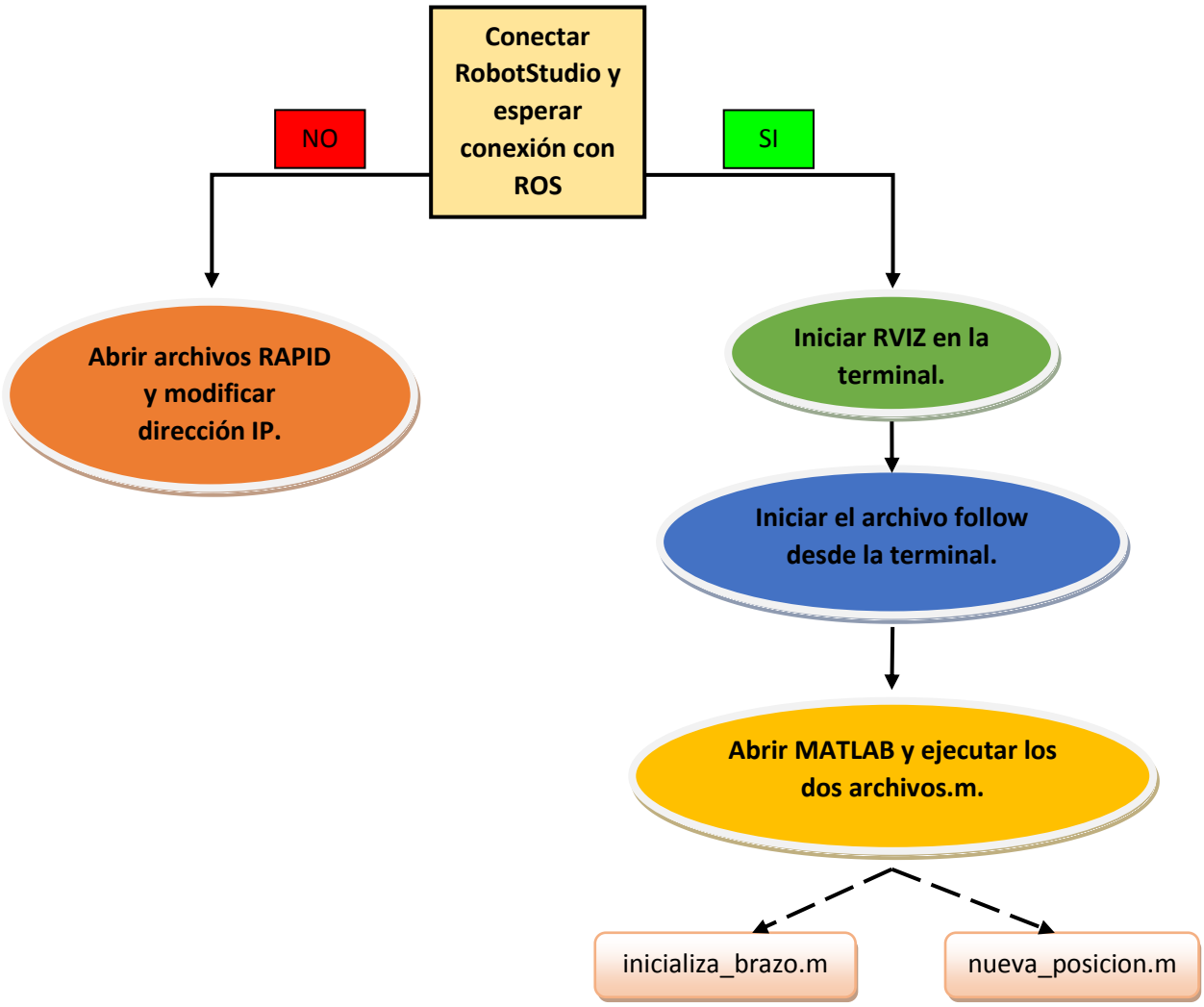
Hay varios proyectos en los que se puede trabajar en un futuro partiendo de este:

- Intentar realizar la comunicación con diferentes tipos de robots.
- Realizar la comunicación en un escenario real, el siguiente paso a la simulación.

Capítulo 8

Planos

Diagrama global



Capítulo 9

Pliego de condiciones

Para este proyecto, son necesarias las siguientes condiciones:

- Un ordenador que utilice GNU/Linux 12.04 con espacio suficiente para poder almacenar los datos utilizados y las simulaciones que se lleven a cabo. De hecho, para un trabajo apropiado se debe de disponer de un ordenador potente. Los siguientes paquetes también son necesarios:
    - ROS – versión completa
    - ROS Industrial
    - MoveIt!
    - MATLAB versión R2015a
- Otro ordenador con RobotStudio y con tarjeta gráfica y memoria suficiente para realizar las simulaciones del brazo robot.

Capítulo 10

Presupuesto

En este capítulo se describe el coste teórico de todo el proyecto.

10.1 Coste material

En esta sección se muestra una tabla con el coste de los diferentes materiales que se han utilizado en este proyecto.

Dispositivos		Precio (euROS)	Unidades	Coste total (euROS)
Hardware	Ordenador HP	650	1	650
	Windows PC	450	1	450
Coste total del Hardware				1100
Software	Ubuntu v12.04		1	0
	ROS		1	0
	RobotStudio		1	0
	MATLAB		1	0
Coste total del software				0
Coste total del material				1100

Tabla 3: Coste material.

10.2 Coste profesional

En esta sección se ha calculado el coste profesional. Incluye todas las actividades profesionales relacionadas con el proyecto.

Actividad	Precio (euROS)	Tiempo (meses)	Coste total (euROS)
Ingeniería	1500	3	4500
Mecanografía	1000	1	1000
Coste total			5500

Tabla 4: Costes profesionales.

Se calcula que cada hora de ingeniería se paga a 50€ hora.

10.3 Coste total

Coste material	1100
Coste profesional	5500
Impresión	70
Transporte	150
Total	6820

Tabla 5: Costes adicionales y totales.

Capítulo 11

Manual de usuario

- En el **ordenador con Windows y Robotstudio**, abrir el proyecto de Robotstudio con la configuración ya creada.
- Acceder a la pestaña "RAPID" y comprobar que la dirección IP que está escrita es la correcta.
- Desplazar el brazo robot para su vuelta a la posición inicial más adelante.
- Abrir la pestaña "FLEX-PENDANT" y posteriormente "VIRTUAL FLEX-PENDANT" y esperar la conexión desde el ordenador con Linux.

- En el **ordenador con Linux**, hay que establecer la comunicación; por lo que se abre un terminal. En él se va a acceder a la carpeta que contiene los archivos de RAPID, se compila y se lanza el programa que establecerá la comunicación. Para ello se escribe lo siguiente:
  - *cd /…/ packages*
  - *catkin_make*
  - *source /…/ packages/ devel/ setup.bash*
  - *roslaunch abb_common robot_interface robot_ip: xxx.xxx.xxx.xxx*
- Una vez establecida la comunicación, se abre otro terminal y se lanza el archivo de MoveIt! para que calcule la trayectoria del brazo robot:
  - *source /…/ packages/ devel/ setup.bash*
  - *roslaunch irb120_moveit_config move_group_fast.launch*
- El siguiente paso es lanzar el archivo que se va a encargar de mandar los puntos a MoveIt!, el archivo follow:
  - *source /…/ packages/ devel/ setup.bash*
  - *roslaunch follow follow*
- En un nuevo terminal, se abre MATLAB. Hay dos opciones, seguir paso a paso las instrucciones de creación de un mensaje para posteriormente enviarlo, o inicializar los dos archivos previamente creados:
  - *rosinit*
  - *rostopic info '/Object_pose'*
  - *posepub=rospublisher('/Object_pose','geometry_msgs/Pose')*
  - *posemsg=rosmessage(posepub)*

- *posemsg.Position.X=0.4*
- *posemsg.Position.Y=-0.3*
- *posemsg.Position.Z=0.05*
- *send(posepub,posemsg)*

- Una vez realizado paso a paso lo anterior, se ha creado un Publisher, se ha creado un mensaje, se ha editado ese mensaje y se ha mandado al nodo encargado del movimiento del brazo robot.
- Por último, al ser enviado el mensaje, MoveIt! calcula la trayectoria y se lo manda al brazo robot, que irá a su posición inicial para posteriormente ir a la posición que se ha escrito en el mensaje enviado.

Bibliografía

"ABB web oficial", http://new.abb.com/

"ROS web oficial", http://www.ROS.org/core-components/

"MoveIT! web oficial", http://moveit.ROS.org/

"OMPL web oficial", http://ompl.kavrakilab.org/

"ROS Industrial web oficial", http://ROSindustrial.org/

"MATLAB web oficial", http://es.mathworks.com

"Instalación de ROS", http://wiki.ROS.org/es/hydro/Installation/Ubuntu

Jorge Nicho, "ROS socket for the ABB robots", http://wiki.ROS.org/abb/Tutorials/InstallServer

"Robot Operating System Support from Robotics System Toolbox", https://es.mathworks.com/hardware-support/robot-operating-system.html

"Get Started with ROS", https://es.mathworks.com/help/robotics/examples/get-started-with-ROS.html

"Exchange data with ROS Publishers and Subscribers", https://es.mathworks.com/help/robotics/examples/exchange-data-with-ROS-publishers-and-subscribers.html

"Work with Basic ROS Messages", https://es.mathworks.com/help/robotics/examples/work-with-basic-ROS-messages.html

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá